Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

EMPIRICAL PAPER

# The Effect of Hoisting on Variants of Hierarchical Delta Debugging

Dániel Vince* | Renáta Hodován | Daniella Bársony | Ákos Kiss

Department of Software Engineering, University of Szeged, Szeged, Hungary

**Correspondence**
*Dániel Vince, Department of Software Engineering, University of Szeged, Szeged, Hungary. Email: vinced@inf.u-szeged.hu

## Abstract

Minimizing failing test cases is an important pre-processing step on the path of debugging. If much of a test case that triggered a bug does not contribute to the actual failure, then the time required to fix the bug can increase considerably. However, test case reduction itself can be a time-consuming task, especially if done manually. Therefore, automated minimization techniques have been proposed, the minimizing Delta Debugging and the Hierarchical Delta Debugging algorithms being the most well known.
In this paper, we investigated the input format of HDD, searching for structures that the algorithm cannot reduce. Motivated by the findings, we have created an algorithmic framework that enabled the use of transformations other than pruning. Furthermore, with the Transformation-based Minimization framework, we propose to extend HDD and its coarse and recursive variants with a reduction method that does not prune subtrees, but replaces them with compatible subtrees further down the hierarchy, called hoisting. We have evaluated various combinations of pruning and hoisting on multiple test suites and found that hoisting can help to further reduce the size of test cases by 27% on average and by 80% as best case compared to the baseline algorithm.

**KEYWORDS:**
test case minimization, hierarchical delta debugging, transformation-based minimization, hoisting

## 1 | INTRODUCTION

If anything can fail, it will fail – our carefully crafted software included. If we are lucky enough, we have a record of the events or inputs that triggered the failure. With some more luck, the failure is reproducible. In this case, a lucky engineer will get the task of fixing the problem.

Usually, the observed problem is only a symptom, and the root of it has to be found first in order to get the bug fixed. The record of the events or inputs that triggered the failure – i.e., the test case – can help here. However, test cases are often a mixture of relevant and irrelevant information, and if much of the test case is irrelevant, i.e., it does not contribute to the failure, then the engineering time and effort required to fix the bug can increase considerably. Therefore, the minimization of failure-inducing test cases is an important first step on the path of debugging. However, it is of limited benefit if the expensive engineer-hours spent on bug fixing are simply spent on manual test case reduction instead of starting the investigation from a minimal input sequence that reproduces the failure.

One of the most well-known techniques to automate reduction is the minimizing Delta Debugging algorithm (DDMIN) by Zeller and Hildebrandt[1,2,3], working on all kinds of inputs without the need for any information about their internal structure. It has been realized, however, that if the input structure is known, that knowledge can be utilized to create smaller results faster. Misherghi and Su have introduced the Hierarchical Delta Debugging (HDD) algorithm[4,5], built on DDMIN, that works on tree-structured inputs (e.g., on any input format that has a context-free grammar) and prunes unnecessary subtrees of the test case during reduction. These foundational studies have inspired many researchers to improve the efficiency[6,7,8,9] and the effectiveness[10] of the algorithms. As a result, several HDD variants have been published, such as HDD$^r$[9] and Coarse HDD[8], which modified the order of tree traversal and the examinable nodes for HDD, respectively.

This paper is an extension of our previous work[11], which proposed to extend HDD, where subtrees of the test case are removed, with a technique called hoisting, where subtrees are replaced with compatible subtrees down the hierarchy. In this follow-up work, we propose to extend three additional HDD variants with hoisting: HDD$^r$, Coarse HDD, and Coarse HDD$^r$. The latter variant is a new combined utilization of Coarse HDD and HDD$^r$, which has not been presented in the literature yet, therefore, we give a formalization for the algorithm. We conduct an exhaustive analysis in order to compare the effect of hoisting on different HDD variants from both an effectiveness and an efficiency points of view. Furthermore, we investigate the tree structure of the preprocessed inputs, whether there are correlations between structure patterns and the effect of our extension. In this work, we also focus on improving the effectiveness of reduction, accordingly, our goal is to answer the following research questions:

Research Questions

**RQ1.** Are there any tree structures that HDD and its variants are not able to minimize, but a human engineer can easily reduce?

**RQ2.** Can tree structure transformations beyond pruning be defined in a way that can be incorporated into HDD algorithm variants?

**RQ3.** Is the effectiveness (i.e., the effect on the size of reduced test cases) of hoisting similar on all HDD variants?

**RQ4.** Is the efficiency (i.e., the effect on the number of testing steps) of hoisting similar on all HDD variants?

**RQ5.** Is there a best-performing HDD variant?

Therefore, we define the algorithmic framework of hoisting and describe its potential combinations with pruning. We have evaluated the introduced approaches and found that hoisting can help to further reduce the size of test cases by as much as 80% compared to the baseline HDD variants.

The rest of the paper is organized as follows: first, in Section 2, we give a brief overview of DDMIN, HDD, and its variants, to make this paper self-contained. Then, in Section 3, we show some motivational examples where pruning-based reduction can be improved upon, and describe and formalize the idea of hoisting. In Section 4, we evaluate the effects of hoisting with the help of a prototype implementation, and we present our experimental results. In Section 5 we discuss related work, and finally, in Section 6 we summarize our work and conclude the paper.

## 2 | BACKGROUND

The minimizing Delta Debugging (DDMIN) algorithm[1,2,3] is a systematic iterative approach for reducing a test case while keeping some interesting property invariant. The algorithm works on a set of atomic units representing parts of the test case. First, this set of units is split into two subsets of roughly equal size, and both subsets are investigated for whether they still have the interesting property of the original test case. If the property is kept in any of the subsets, then reduction was successful and a new iteration starts with the found subset, otherwise, the granularity is refined by doubling the splitting. The subsets of the new partitioning are investigated again, one by one, as well as their complements, i.e., it is checked whether keeping or removing any of the subsets leads to an interesting smaller test case. Again, if any of the investigated test cases keeps the property in question, it will be used as the input for the next iteration, otherwise, the granularity is increased. The iteration continues until the granularity reaches unit-level, when it is proven to have found a so-called 1-minimal result, a local minimum where the removal of any single unit from the test case causes the loss of the interesting property.

The algorithm has its roots in the isolation of failure-inducing code changes, which is visible in its terminology. For the algorithm, a test case is composed of elementary changes or deltas, denoted as $\delta_i$, whence the algorithm got its name. A set of changes is also called a configuration, usually denoted by $c$. The outcome of a program run on a specific configuration is determined by a testing function, and it can be either *fail* (also written as ✗) if the test case induced the original failure, *pass* (also written as ✓) if the test succeeds, or *unresolved* (written as **?**) if the result is indeterminate. The set of all changes, i.e., the initial configuration that triggers a failing run is denoted by $c_{✗}$. Although the algorithm is often applied to the simplification of program inputs where the term "change" is not an intuitive fit to the units of a test case (e.g., to characters or lines of a text file) and the algorithm also has use cases where the interesting property of a test case is not a program failure, most authors, including us, follow the original notation for historical reasons. For the sake of completeness, Figure 1 gives Zeller and Hildebrandt's latest formulation of the minimizing Delta Debugging algorithm[3].

If a test case that is to be reduced has some mandatory structure over its units, which is quite typical for inputs to a program, DDMIN may work suboptimally. The configuration partitioning during the iterations may be completely unaligned with the boundaries of the structural elements of the input, leading to incorrectly formatted, non-reproducing, and thus useless test cases. The goal of the Hierarchical Delta Debugging (HDD)

Let $test$ and $c_{✗}$ be given such that $test(\emptyset) = ✓ \wedge test(c_{✗}) = ✗$ hold.
The goal is to find $c'_{✗} = ddmin(c_{✗})$ such that $c'_{✗} \subseteq c_{✗}$, $test(c'_{✗}) = ✗$, and $c'_{✗}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{✗}) = ddmin_2(c_{✗}, 2) \text{ where}$$

$$ddmin_2(c'_{✗}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = ✗ \text{ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = ✗ \text{ (“reduce to complement”)} \\ ddmin_2(c'_{✗}, \min(|c'_{✗}|, 2n)) & \text{else if } n < |c'_{✗}| \text{ (“increase granularity”)} \\ c'_{✗} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{✗} - \Delta_i$, $c'_{✗} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all $\Delta_i$ are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✗}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✗}) = ✗ \wedge n \leq |c'_{✗}|$.

**FIGURE 1** The Minimizing Delta Debugging algorithm.

algorithm[4] is to avoid such superfluous steps by not testing format-breaking configurations. To achieve that goal, it works on hierarchical tree-structured input representations (e.g., on parse trees, abstract syntax trees, or XML DOM trees) and applies the minimizing Delta Debugging algorithm to the levels of a tree, progressing downwards from the root to the leaves.

The pseudocode formulation of HDD as defined by Misherghi and Su[4] is shown in Figure 2(a). In the algorithm, the auxiliary routine *tagNodes* collects the nodes at a given level of the tree, then DDMIN is invoked on those nodes, and finally, *prune* applies the result of Delta Debugging to the tree. I.e., for HDD, configurations are sets of tree nodes at a given level, and removal of a node causes the removal of the whole subtree rooted at that node. In a later variant of HDD, “pruning” of a node has been reinterpreted as its replacement with the minimal applicable syntactically correct fragment to reduce the number of test attempts at incorrectly formatted configurations even further[5]. If HDD is iterated until a fixed-point is reached, denoted as HDD*, it gives a 1-tree-minimal result, i.e., if any single node is removed from the tree, the new test case will not be interesting anymore.

Since the original definition of HDD, several variants have been proposed, of which we highlight two: the recursive and the coarse hierarchical delta debugging algorithms, or HDD$^{r}$[1] and Coarse HDD for short. The idea behind HDD$^{r}$[9] is to pass only related parts of the tree to DDMIN to ensure that it does not create partitions that cross subtree boundaries (which often leads to superfluous steps). Therefore, HDD$^{r}$ applies DDMIN not to all nodes at a given level of the tree but to sibling nodes only. The intuitive formalization of the idea is of a recursive nature, which gives the name of the algorithm. An alternative iterative formulation also exists for HDD$^{r}$, which is shown in Figure 2(b). The auxiliary routines *tagChildren* and *pruneChildren* differ from their *tagNodes* and *prune* counterparts only in the set of nodes they work on, i.e., on the children of a given node instead of all nodes at a given level. Importantly, HDD$^{r}$ has the same theoretical minimality guarantees as HDD.

Coarse HDD[8] focuses on those parts of the tree that have an empty minimal applicable replacement (which often occurs in parse trees built from extended context-free grammars utilizing quantifiers). The idea is that the effectively complete removal of such subtrees should bring the biggest gain in terms of test case reduction, while other parts of the tree deserve less attention. Therefore, Coarse HDD, as shown in Figure 2(c), visits all levels of the tree like the original HDD, but filters out those nodes from the input configuration of DDMIN that have a non-empty replacement fragment. The auxiliary routine *filterEmptyPhiNodes* performs the above-described filtering (where *Phi* refers to the minimal applicable replacements, which were originally denoted by $\Phi$[5]). Obviously, Coarse HDD reduces test cases without guarantees for theoretical minimality. However, in exchange for the potentially bigger results, it is expected to yield results in fewer steps than HDD. In practice, Coarse HDD can be a preprocessing step as filtering out nodes that have an empty minimal applicable replacement, then the main HDD algorithm minimizes the preprocessed tree faster. We have to note that there is a potential variant of the HDD algorithm that has not been formally defined in the literature, which is the combination of the coarse and recursive ideas, using the recursive iteration approach while focusing on particular nodes only. Defining new HDD variants is not in the direct focus of this paper, however, for the sake of completeness, the algorithm is formally defined in Figure 2(d).

[1]Previous works have used various notations for the recursive HDD algorithm: in some cases, the letter ‘r’ was typeset in small capital (i.e., HDDR), while in others simply in lowercase (i.e., HDDr). Here, for the sake of consistency with latter parts of this paper, we use ‘r’ in a superscript position.

```
procedure HDD($input\_tree$)
    $level \leftarrow 0$
    $nodes \leftarrow tagNodes(input\_tree, level)$
    while $nodes \neq \emptyset$ do
        $minconfig \leftarrow$ DDMIN($nodes$)
        $prune(input\_tree, level, minconfig)$
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

(a)

```
procedure HDD$^r$($root\_node$)
    $queue \leftarrow \langle root\_node \rangle$
    while $queue \neq \langle\rangle$ do
        $current\_node \leftarrow pop(queue)$
        $nodes \leftarrow tagChildren(current\_node)$
        $minconfig \leftarrow$ DDMIN($nodes$)
        $pruneChildren(current\_node, minconfig)$
        $append(queue, minconfig)$
    end while
end procedure
```

(b)

```
procedure CoarseHDD($input\_tree$)
    $level \leftarrow 0$
    $nodes \leftarrow tagNodes(input\_tree, level)$
    while $nodes \neq \emptyset$ do
        $nodes \leftarrow filterEmptyPhiNodes(nodes)$
        if $nodes \neq \emptyset$ then
            $minconfig \leftarrow$ DDMIN($nodes$)
            $prune(input\_tree, level, minconfig)$
        end if
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

(c)

```
procedure CoarseHDD$^r$($root\_node$)
    $queue \leftarrow \langle root\_node \rangle$
    while $queue \neq \langle\rangle$ do
        $current\_node \leftarrow pop(queue)$
        $nodes \leftarrow tagChildren(current\_node)$
        $nodes \leftarrow filterEmptyPhiNodes(nodes)$
        if $nodes \neq \emptyset$ then
            $minconfig \leftarrow$ DDMIN($nodes$)
            $pruneChildren(current\_node, minconfig)$
        end if
        $append(queue, tagChildren(current\_node))$
    end while
end procedure
```

(d)

**FIGURE 2** (a) The Hierarchical Delta Debugging algorithm, (b) the iterative formulation of the Recursive Hierarchical Delta Debugging algorithm, (c) the Coarse Hierarchical Delta Debugging algorithm, and (d) the Coarse Recursive Hierarchical Delta Debugging algorithm.

## 3 | HOISTING

Although HDD and its variants perform better on structured inputs than DDMIN, there is still room for improvement. Several improvements have already been proposed, often by preprocessing the tree representation HDD is working on, e.g., by hiding some tokens from HDD and DDMIN to reduce the number of nodes that have to be considered, by collapsing (a.k.a. squeezing) multiple nodes into one for the same reason[7], or by rotating recursive structures of the tree to reduce its height[8]. However, these transformations do not change the core structure of the tree, i.e., the test case generated (or, serialized) from the preprocessed tree will still be the same as the original input. Because of this, and because of the way all HDD variants work, an HDD-reduced test case – even if 1-tree-minimal – may contain structural elements that a human expert would still remove.

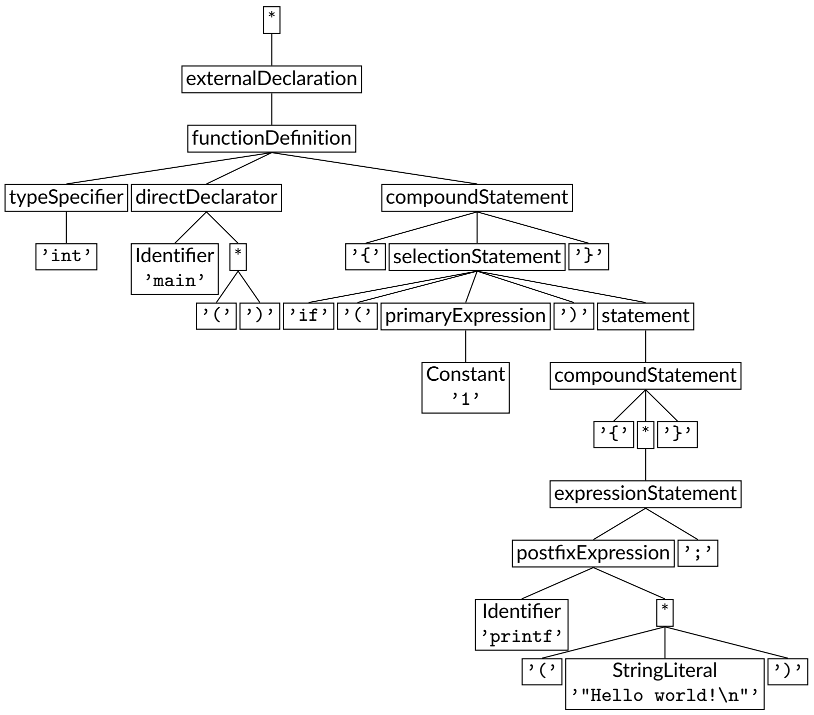
### 3.1 | Examples

A simple example of this suboptimal structure-preserving behavior is shown in Figure 3. The C program in Figure 3(a) prints the classic "Hello world!" message, the printing is wrapped in an *if* statement where the predicate always evaluates to true. If we take this program as a test case and define the printing of the "Hello world!" message as interesting, then we can try and minimize it[2]. Figure 3(b) shows the parse tree of the program, generated by a parser using a context-free grammar of the C programming language and preprocessed for compactness (most notably, squeezing[7] and recursion flattening[8] have been applied). Unfortunately, none of the algorithms presented in Section 2 can reduce this test case any further

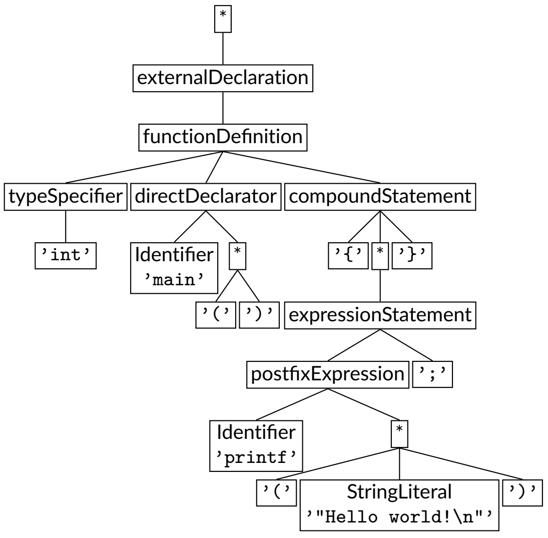
[2]This is an example where the interesting property of the test case is *not* a program failure.

```
int main() {
  if (1) {
    printf("Hello world!\n");
  }
}
```

(a)



(b)



(c)

```
int main() {
  printf("Hello world!\n");
}
```

(d)

**FIGURE 3** An overly complicated "Hello World" program: (a) written in C, (b) its parse tree, (c) its parse tree minimized with hoisting applied to keep printing the "Hello world!" message, and (d) the C program serialized from the minimized tree.

(especially if replacement with a minimal syntactically correct fragment is used when pruning subtrees) as removing any of the nodes (or lines, or characters) would either yield a syntactically incorrect test case, or one that does not print the message, making it uninteresting.

Theoretically, both HDD and HDD$^r$, and also the underlying DDMIN algorithm could be modified to give $n$-(tree-)minimal results, but that would lead to exponential complexity, which is impractical. Thus, we propose another approach called *hoisting*.

We can observe that there are recurring structures in the parse tree, subtrees rooted at nodes with identical labels, denoting the derivation of the same non-terminal of the grammar. The assumption of hoisting is that one such subtree may be replaced with another without losing syntactic correctness, and that subtrees whose roots are in an ancestor-descendant relationship may be good candidates for reduction. Of course, the testing function has to confirm (or reject) whether such a transformation keeps the resulting test case interesting.

In the tree of Figure 3(b), there is one pair of such subtrees, those rooted at nodes labeled as *compoundStatement*. Figure 3(c) shows a transformed tree where the descendant subtree is hoisted to replace all the structures that enclosed it. When this tree is serialized into the form of a C program (see Figure 3(d)), it becomes apparent that, in this case, this transformation was indeed useful and we got a smaller and still interesting test case.

When discussing the idea of hoisting with fellow researchers, the argument was often raised that such a transformation is only good for removing some minor syntactic elements from the result, like a dangling semicolon or a pair of superfluous braces, etc. The example in Figure 3 does not seem to contradict such arguments. However, Figure 4(a) shows another example, a program written in Java, that prints the rounded value of $\pi$ in a localized format, provided that the specified locale is supported, and throws an exception otherwise.

As presented, the program only supports the *en* locale. We shall take this program as a test case and the testing function shall check whether the program exits without error with parameter *en* and throws an exception when invoked with an unsupported locale (e.g., *hu*). If HDD is used to minimize this test case, it will be able to remove some parts of the program, but most of the original structure will remain in the output. (Because the exception that needs to be thrown is in the *decSeparator* method, which is called inside a call to the *formatParts* method, both methods are forced to be kept in the reduced test case.) The parse tree for this program would be too big to be presented as an example, so we only show the

```
public class LocalizedPi {
  private static String decSeparator(String locale) {
    if (locale.equals("en")) {
      return ".";
    }
    throw new Exception("Unsupported locale");
  }
  private static String formatParts(String intPart, String fracPart, String decSep) {
    return intPart.concat(decSep).concat(fracPart);
  }
  public static void main(String[] args) {
    String pi = formatParts("3", "14", decSeparator(args[0]));
    System.out.println(pi);
  }
}
```

(a)

```
class LocalizedPi {
  static String decSeparator(String locale) {
      if (locale.equals("en")) {
          return "";
      }
      throw new Exception("Unsupported locale");
  }
  static String formatParts(String intPart, String fracPart, String decSep) {
      return decSep;
  }
  public static void main(String[] args) {
      String a = formatParts("", "", decSeparator(args[0]));
  }
}
```

(b)

```
class LocalizedPi {
  static String decSeparator(String locale) {
      if (locale.equals("en")) {
          return "";
      }
      throw new Exception("Unsupported locale");
  }
  public static void main(String[] args) {
      String a = decSeparator(args[0]);
  }
}
```

(c)

**FIGURE 4** A program to print the rounded value of $\pi$ in a locale-specific format: (a) written in Java, (b) minimized with HDD to keep the program throwing an uncaught exception if an unsupported locale is specified on the command line, and (c) minimized with hoisting applied before HDD.

HDD-reduced Java program in Figure 4(b), which also displays precisely what HDD can and what it cannot prune away. However, if hoisting is used *before* HDD, it can pave the way for the latter reduction technique by hoisting the call to *decSeparator* to replace the enclosing call to *formatParts*, thus allowing the complete removal of the definition of *formatParts*. (In this example, the method calls within *main* are the recurring structures that are in ancestor-descendant relation in the tree.) The result of the combined application of the two techniques is presented in Figure 4(c).

Note that *formatParts* could be constructed arbitrarily complex, making the theoretical potential of hoisting considerably higher than the removal of anecdotical semicolons or curly braces. Also note, however, that hoisting cannot achieve this alone, as it "only" moves subtrees higher up the tree, but it has to cooperate with HDD.

## 3.2 | Transformation-based Minimization

To formalize the ideas motivated and described above, we built on and extended the notations and terminology of delta debugging (as given in Section 2) to introduce transformation-based minimization[11] first. In the context of delta debugging, a test case is always composed of a subset that contains elements of the initial failing configuration. The testing function is also defined for the subsets of $c_{\text{✗}}$ only. However, it is useful (actually, necessary) if we can also determine the outcome of a program run on a set of elements, even if some of them are not part of the initial configuration

Let $D$ denote the set of all potential test case elements, and let $\delta \in D$ denote one element of that set.
Let $test$ and $c_{✗} = \{\delta_1, \ldots, \delta_n\} \subseteq D$, and $test(c_{✗}) = ✗$ holds.
Let $\tau$ and $\|\cdot\|$ be given such that $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds.
The goal is to find $t_{✗} = tmin^{\tau}(c_{✗})$ such that $test(\bar{t}_{✗}(c_{✗})) = ✗$ and $t_{✗}$ is 1-maximal.
The *transformation-based minimizing algorithm* $tmin^{\tau}(c)$ is

$$tmin^{\tau}(c_{✗}) = tmin_2^{\tau}(c_{✗}, id_D) \text{ where}$$

$$tmin_2^{\tau}(c_{✗}, t'_{✗}) = \begin{cases} tmin_2^{\tau}(c_{✗}, t'_{✗}[\delta \mapsto \delta']) & \text{if } \exists \delta \in c_{✗} \cdot \exists \delta' \in \tau(t'_{✗}(\delta)) \cdot test(\bar{t'_{✗}}[\delta \mapsto \delta'](c_{✗})) = ✗ \\ t'_{✗} & \text{otherwise.} \end{cases}$$

The recursion invariant (and thus precondition) for $tmin_2^{\tau}$ is $test(\bar{t'_{✗}}(c_{✗})) = ✗$.

**FIGURE 5** The Transformation-based Minimizing Algorithm.

(e.g., in the case of hoisting, when an element (a node) is replaced by another element (another node further down the hierarchy), which is part of the tree, but is not a member of the initial set). Therefore, we generalize the definitions of delta debugging[3] as follows.

Let $D$ denote the set of all potential test case elements, and let $\delta \in D$ denote one element of that set, i.e., a test case element. A test case or configuration is denoted as $c \subseteq D$. A testing function $test : 2^D \to \{✗, ✓, ?\}$ shall determine for any test case whether it produces the failure in question. The initial failing configuration is denoted as $c_{✗} = \{\delta_1, \ldots, \delta_n\} \subseteq D$, and $test(c_{✗}) = ✗$ holds.

As $c_{✗}$ is a subset of a potentially larger set $D$, we allow for *transformations* that can not only remove, but also *replace* elements in the configuration. We use the following definitions and notations for transformations:

A function $t : D \to D$ is a transformation of test case elements, and the identity transformation is $id_D : D \to D; \delta \mapsto \delta$. We also define the application of a transformation to test cases (or configurations) as $\bar{t} : 2^D \to 2^D; c \mapsto \{t(\delta) : \delta \in c\}$ (e.g., $\overline{id}_D(c_{✗}) = c_{✗}$). And a transformation that is derived from another transformation by changing the mapping of one test case element is defined as

$$t[\delta' \mapsto \delta''] : D \to D; \delta \mapsto \begin{cases} \delta'' & \textit{if } \delta = \delta' \\ t(\delta) & \textit{otherwise.} \end{cases}$$

In the examples of Section 3.1, the transformations that could be applied were quite straightforward. There was only one *compoundStatement* and one method call that could potentially replace their parents. In a general case, however, a test case element may have multiple replacement candidates (or none at all). This is formalized using a function $\tau : D \to 2^D$ that maps test case elements to their transformed candidates.

Finally, as test cases are not necessarily subsets of the initial failing configuration, minimality cannot be defined in terms of the subset relation anymore. Thus, we expect a $\|\cdot\|$ measure to exist on set $D$. If all transformation candidates in $\tau$ are potentially reducing the size of a configuration according to the measure $\|\cdot\|$, i.e., $\forall \delta \in D \cdot \forall \delta' \in \tau(\delta) \cdot \|\delta'\| < \|\delta\|$ holds, then in order to minimize the test case we have to maximize the replacements applied to the elements of the initial configuration (even transitively) while ensuring that the so-transformed test case remains interesting. Just like it is true for DDMIN that searching for the global optimum is impractical, so is it also true for transformation-based minimization. Therefore, our actual goal is to find a local optimum, a *1-maximal* transformation $t_{✗}$ such that $\forall \delta \in c_{✗} \cdot \forall \delta' \in \tau(t_{✗}(\delta)) \cdot test(\bar{t}_{✗}[\delta \mapsto \delta'](c_{✗})) \neq ✗$ holds.

Figure 5 wraps up this subsection and formalizes the transformation-based minimizing algorithm TMIN$^{\tau}$, worded in the likeness of DDMIN.

## 3.3 | Variants of Hoisting and HDD

The transformation-based minimizing algorithm gives us a framework to formulate hoisting. The key to this is to define hoisting as a transformation of tree nodes. More precisely, to define those nodes in the tree representation of the input that can act as replacement candidates for their ancestors. The formula in Figure 6, $\chi(n)$, is one possible way to define these candidates, i.e., the hoistable descendants of a node $n$. $\chi(n)$ is given in terms of two auxiliary functions, of which *children*($n$) is trivial, giving the direct descendants of a node, whereas *compatible*($n, n'$) leaves some space for interpretation. In an extreme case, any two nodes could be considered compatible, but that would rarely be useful. If the tree representation of the input is built using a context-free grammar as motivated in Section 3.1, then a natural interpretation is to regard identically-labeled nodes (i.e., subtrees of derivations of the same non-terminal of the grammar) as compatible.

A basic measurement for nodes of a tree is based on the size of their subtrees, i.e., the number assigned by the measurement to a node $n$ equals the number of nodes in the subtree of $n$. It is obvious that all transformation candidates returned by $\chi(n)$ reduce the size of the configuration according to this measure, as expected by the definition of TMIN.

$$\chi(n) = \bigcup_{n' \in children(n)} \chi'(n, n')$$

$$\chi'(n, n') = \begin{cases} \{n'\} & \text{if } compatible(n, n') \\ \bigcup_{n'' \in children(n')} \chi'(n, n'') & \text{otherwise} \end{cases}$$

**FIGURE 6** $\chi(n)$, the potentially hoistable descendants of node $n$.

Now, with the help of TMIN$^\chi$, we can introduce a hierarchical algorithm, called Hoist, that works its way through the tree from the root to the leaves, and uses TMIN$^\chi$ to find the hoisting transformations at each level. Candidates found by TMIN$^\chi$ are prioritized by their distance to the ancestor, further nodes getting higher priority. The pseudocode of the algorithm is presented in Figure 7(a). The structure of Hoist is similar to that of HDD: both contain a loop to iterate through the levels of a tree, and inside the loop, both perform a minimization step (TMIN$^\chi$ vs. DDMIN) and the application of its result to the tree (via the *transform* and *prune* auxiliary functions).

As discussed at the example of Figure 4, Hoist can achieve reduction on its own, although it is expected to work best if used in combination with HDD, e.g., by using Hoist as a preprocessing step. However, inspired by the similarities between the variants of these two algorithms, we can think of other ways to combine them as well. E.g., the bodies of the loops can be interlaced, performing both the DDMIN and TMIN$^\chi$-based minimizations at each iteration. One way to formulate this idea is shown in Figure 8, where HDD and Hoist are interlaced in the algorithm named HDDH.

In our previous work, we have investigated the effect of Hoist with the original HDD algorithm, as introduced by Misherghi and Su[4,5]. However, because of the similarities between HDD variants and the Hoist algorithm, a recursive, a coarse, and a coarse recursive variant of the hoisting algorithm can also be defined. These are given in Figures 7(b), 7(c), and 7(d), and are named Hoist$^r$, Coarse Hoist, and Coarse Hoist$^r$, respectively. Similarly, we can create new combined algorithms from HDD$^r$, and Hoist$^r$ (i.e., HDDH$^r$), from Coarse HDD, and Coarse Hoist (i.e., Coarse HDDH) and from HDD$^r$ and Coarse Hoist (i.e., Coarse HDDH$^r$). These combinations are trivial following the example of HDDH, therefore, they are not shown to avoid unnecessary repetition.

## 4 | EXPERIMENTAL RESULTS

### 4.1 | Experiment Setup

To evaluate the effects of hoisting, we have implemented the algorithms listed above in the open-source Picireny project[3]. Picireny is a hierarchical test case reduction framework written in Python that supports ANTLR v4[4] grammars and already contains an implementation of the HDD algorithm and its HDD$^r$ and Coarse HDD variants. In our implementation of hoisting, we consider nodes labeled with the same non-terminal of the grammar as compatible, as discussed in Section 3.3.

As inputs, we have collected test cases from different sources, all of which have already been used in the literature for benchmarking reduction. The first test suite is the Perses Test Suite[5], which contains fuzzer-generated C sources that cause various internal compiler errors in the Clang and GCC compilers[6]. The second test set is the JerryScript Reduction Test Suite[7] (JRTS), which contains fuzzer-generated JavaScript files that cause failures in the JerryScript lightweight JavaScript engine. In the case of both test suites, the interesting property of the test cases to keep during reduction is the failure they induce.

For each test case, we built its parse tree representation using the grammar available for the input format from the official ANTLR v4 grammars repository[8]. Before evaluating any of the reduction algorithm combinations, we have applied the squeezing of linear tree components[7] and the flattening of recursive structures to the trees[8]. The properties of the test cases are shown in Table A1 of Appendix A. *Size* is expressed as the number of non-whitespace characters in the test case, *Tree Height* represents the height of the squeezed and flattened parse tree built from the input, *Rules*

[3]https://github.com/renatahodovan/picireny
[4]https://github.com/antlr/antlr4
[5]https://github.com/perses-project/perses
[6]The Perses Test Suite comes with a docker environment provided by its authors. The environment is presumed to contain all compiler versions and tools required to reproduce the issue of each test case in the suite. However, that turned out not to be the case in practice. Thus, we have only used those test cases for evaluation that worked as expected at the time of carrying out the experiments of this paper.
[7]https://github.com/vincedani/jrts
[8]https://github.com/antlr/grammars-v4

```
procedure Hoist($input\_tree$)
    $level \leftarrow 0$
    $nodes \leftarrow tagNodes(input\_tree, level)$
    while $nodes \neq \emptyset$ do
        $hoisting \leftarrow \mathrm{TMIN}^{\chi}(nodes)$
        $transform(input\_tree, level, hoisting)$
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

(a)

```
procedure Hoist$^{\mathrm{r}}$($root\_node$)
    $queue \leftarrow \langle root\_node \rangle$
    while $queue \neq \langle\rangle$ do
        $current\_node \leftarrow pop(queue)$
        $nodes \leftarrow tagChildren(current\_node)$
        $hoisting \leftarrow \mathrm{TMIN}^{\chi}(nodes)$
        $transformChildren(current\_node, hoisting)$
        $append(queue, tagChildren(current\_node))$
    end while
end procedure
```

(b)

```
procedure CoarseHoist($input\_tree$)
    $level \leftarrow 0$
    $nodes \leftarrow tagNodes(input\_tree, level)$
    while $nodes \neq \emptyset$ do
        $nodes \leftarrow filterEmptyPhiNodes(nodes)$
        if $nodes \neq \emptyset$ then
            $hoisting \leftarrow \mathrm{TMIN}^{\chi}(nodes)$
            $transform(input\_tree, level, hoisting)$
        end if
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

(c)

```
procedure CoarseHoist$^{\mathrm{r}}$($root\_node$)
    $queue \leftarrow \langle root\_node \rangle$
    while $queue \neq \langle\rangle$ do
        $current\_node \leftarrow pop(queue)$
        $nodes \leftarrow tagChildren(current\_node)$
        $nodes \leftarrow filterEmptyPhiNodes(nodes)$
        if $nodes \neq \emptyset$ then
            $hoisting \leftarrow \mathrm{TMIN}^{\chi}(nodes)$
            $transformChildren(current\_node, hoisting)$
        end if
        $append(queue, tagChildren(current\_node))$
    end while
end procedure
```

(d)

**FIGURE 7** (a) The Hoisting algorithm, (b) the recursive Hoisting algorithm, (c) the coarse Hoisting algorithm, and (d) the recursive coarse Hoisting algorithm.

show the number of non-terminals, and *Tokens* show the number of terminals in it. The fuzzer-generated C sources are an order of magnitude bigger than the JavaScript files, both in terms of character count and in their internal representation. This diverse input selection provides a wide spectrum to properly investigate the effects of hoisting on differently structured and sized inputs. The experiments were executed on a workstation equipped with an Intel Core i5-9400 CPU clocked at 2.9 GHz and 16 GB RAM. The machine was running Ubuntu 20.04 with Linux kernel 5.4.0.

## 4.2 | The Effects of Hoisting

In our previous work[11], we investigated the effect of hoisting on the HDD algorithm and compared the results with the publicly available versions of the Perses[12] and Pardis[13] tools. We have already ruled out Pardis in that discussion, since Perses gave smaller results in every case of that experiment. Furthermore, in 15 of 19 inputs of the Perses Test Suite, at least one of the hoisting variants gave smaller results than Perses, and in 11 of 19, all of them did. Plus, in all of the 13 inputs of the JRTS, the hoisting-extended HDD algorithm outperformed the Perses tool in terms of reducing JavaScript files. Motivated by these previous results, only the above-discussed HDD algorithm variants are included in this discussion, and the reader is referred to the previous paper if interested in the comparison with the Perses and Pardis tools.

We have conducted four experiments, one for each variant of HDD and hoisting (i.e., original, recursive, coarse, and coarse recursive). For each variant, we used four different combinations of HDD and hoisting as summarized in Table 1. The columns are organized as follows:

- **Algorithm:** the fixed-point iteration of the variant without any hoisting (e.g., HDD*) acted as the baseline,
- **Preprocessing:** hoisting was applied as a preprocessing step to hierarchical delta debugging (e.g., Hoist*+HDD*),

```
procedure HDDH($input\_tree$)
    $level \leftarrow 0$
    $nodes \leftarrow tagNodes(input\_tree, level)$
    while $nodes \neq \emptyset$ do
        $minconfig \leftarrow$ DDMIN($nodes$)
        $prune(input\_tree, level, minconfig)$
        $hoisting \leftarrow$ TMIN$^{X}$($minconfig$)
        $transform(input\_tree, level, hoisting)$
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

**FIGURE 8** The Hierarchical Delta Debugging & Hoisting algorithm.

- **Interlacing:** hoisting was interlaced with hierarchical delta debugging (e.g., HDDH*), and
- **Preprocessing & Interlacing:** as stand-alone hoisting and the interlaced algorithm are not mutually exclusive, they can be used in sequence (e.g., Hoist*+HDDH*).

**HDD.** To make this paper self-contained, the discussion of HDD is also included in the experimental results, despite the fact that this is not the main contribution of this study. In the first experiment, we compared HDD*, Hoist*+HDD*, HDDH*, and Hoist*+HDDH* (see Figures 2(a), 7(a), and 8). On the Perses Test Suite, all hoisting-based algorithm combinations produced a smaller output than the baseline in 17 of 19 cases. (For one input, *gcc-70127*, HDD* ran out of memory, but the combinations with hoisting correctly finished the minimization. To avoid a biased interpretation of data, we do not consider the correctly performing variants better than the baseline in this case.) The reduced test cases could be up to 80.27%, 69.59%, and 80.63% smaller (using Hoist*+HDD*, HDDH*, and Hoist*+HDDH*, respectively) than the result of HDD*. The average effect on size was 37.15%, 38.54%, and 39.82%, respectively. When comparing hoisting combinations to each other, HDDH* gave the smallest result in 8 cases, Hoist*+HDDH* produced the smallest output in 9 cases, while there was also a tie, where HDDH* and Hoist*+HDDH* produced (exactly) the same output. Furthermore, there was another tie when all three approaches found the same local minimum (*gcc-71626*). On the 13 inputs of the JRTS, all algorithms worked quite similarly with respect to the output size: there were many cases where some or all approaches gave identical results. Still, in 10 cases, all hoisting-based approaches gave strictly smaller output than the baseline (by 50%, 47.14%, and 50% in the best cases), while in the other cases none of them gave worse results. On this test suite, the average improvement of the approaches over HDD* was 12.85%, 12.63%, and 12.85%, respectively.

Regarding efficiency, we found that hoisting could have a positive effect on the number of overall test case evaluations, but not necessarily. HDDH* performed the minimization of 17 inputs faster than the baseline HDD*, but in those approaches where hoisting was a preprocessing step, this improvement was only visible in 13 cases. However, JRTS gave significantly different results efficiency-wise than the Perses Test Suite. In the vast majority of cases, the application of hoisting increased the number of testing steps performed during reduction. Hoist*+HDD*, HDDH*, and Hoist*+HDDH* were slower than HDD* in 12, 8, and 13 of the 13 cases, respectively. The raw data for this experiment are shown in Table A2 of Appendix A.

**TABLE 1** Hoisting and the variants of HDD

| **Algorithm** | **Preprocessing** | **Interlacing** | **Preprocessing & Interlacing** |
|---|---|---|---|
| HDD* | Hoist* + HDD* | HDDH* | Hoist* + HDDH* |
| HDD$^{r*}$ | Hoist$^{r*}$ + HDD$^{r*}$ | HDDH$^{r*}$ | Hoist$^{r*}$ + HDDH$^{r*}$ |
| Coarse HDD* | Coarse Hoist* + HDD* | Coarse HDDH* | Coarse Hoist* + HDDH* |
| Coarse HDD$^{r*}$ | Coarse Hoist$^{r*}$ + HDD$^{r*}$ | Coarse HDDH$^{r*}$ | Coarse Hoist$^{r*}$ + HDDH$^{r*}$ |

**HDD$^r$.** In the second experiment, the recursive HDD variant and its combinations with hoisting have been investigated, i.e., HDD$^{r*}$, Hoist$^{r*}$+HDD$^{r*}$, HDDH$^{r*}$, and Hoist$^{r*}$+HDDH$^{r*}$ (see Figures 2(b) and 7(b)). On the Perses Test Suite, the interlaced application of hoisting (HDDH$^{r*}$) resulted in smaller outputs than the baseline HDD$^r$ for all tests by 42.4% on average. Improvement was observed in 17 of 19 cases when using Hoist$^{r*}$+HDD$^{r*}$ (41.12% on average) and in 18 of 19 cases when using Hoist$^{r*}$+HDDH$^{r*}$ (43.71% on average). Reduced inputs from the JRTS also became smaller, the average improvement was 12.17%, 15.2%, and 15.79% on average compared to the baseline.

On the Perses Test Suite, the hoisting-extended combinations required fewer testing steps by 9.8%, 27%, and 14.16% on average compared to the baseline HDD$^{r*}$. The analysis of efficiency shows a similar pattern to effectiveness: HDDH$^{r*}$ reduced all of the inputs faster than the baseline, while this boost was only observable in 15 of 19 cases with Hoist$^{r*}$+HDD$^{r*}$ and 14 of 19 cases with Hoist$^{r*}$+HDDH$^{r*}$. Furthermore, if the results are compared to the traditional HDD*, Hoist$^{r*}$+HDD$^{r*}$, HDDH$^{r*}$, and Hoist$^{r*}$+HDDH$^{r*}$ required 70.17%, 75.33%, and 71.6% fewer steps on average, respectively. However, when investigating efficiency on JRTS, we got somewhat different results: HDDH$^{r*}$ required 8% more testing steps, while preprocessing the parse-tree with hoisting roughly doubled the reduction effort. Data for this experiment are shown in Table A3.

**Coarse HDD.** In the third experiment, the Coarse HDD variants have been investigated, i.e., Coarse HDD*, Coarse Hoist*+HDD*, Coarse HDDH*, and Coarse Hoist*+HDDH* (see Figures 2(c) and 7(c)). When hoisting acted as a preprocessing step, algorithm combinations produced smaller output than the baseline in all cases on the Perses Test Suite. The output of the reduction became smaller by 53.06% and 53.07% than the baseline (using Coarse Hoist*+HDD* and Coarse Hoist*+HDDH*, respectively). Coarse HDDH* produced exactly the same output as the baseline, except in two test cases, where the outputs were negligibly larger (by 2%). On the JRTS, hoisting as a preprocessing step gave 15% smaller outputs compared to the baseline, and Coarse HDDH* produced the same output as the baseline.

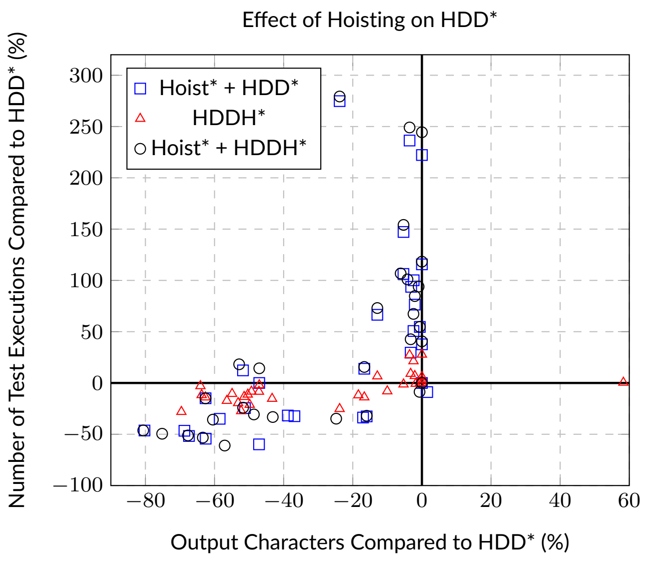
In Coarse HDD[8], the 1-minimality guarantee is sacrificed to speed up the reduction process. Efficiency-wise, using Coarse Hoist*+HDD* and Coarse Hoist*+HDDH* variants, the number of test executions has increased heavily, while the Coarse HDDH*executed the reduction exactly the same way as the baseline Coarse variant. It can be concluded from the experimental results that hoisting is effective only as a preprocessing step if the main reduction algorithm is the Coarse HDD. The backing data for this experiment are shown in Table A4.

**Coarse HDD$^r$.** In the last experiment, coarse recursive variants have been compared, i.e., Coarse HDD$^{r*}$, Coarse Hoist$^{r*}$+HDD$^{r*}$, Coarse HDDH$^{r*}$, and Coarse Hoist$^{r*}$+HDDH$^{r*}$ (see Figures 2(d) and 7(d)). Table A5 contains the results for both test suites, which show quite a few similarities to the non-recursive Coarse variant. The reduction produced smaller test cases by 54.37% and 53.19% than the baseline on Perses Test Suite and 11.05% and 11.05% on JRTS (using Coarse Hoist$^{r*}$+HDD$^{r*}$ and Coarse Hoist$^{r*}$+HDDH$^{r*}$, respectively), however, Coarse HDDH$^{r*}$ produced larger C outputs in three cases by 6.56%.

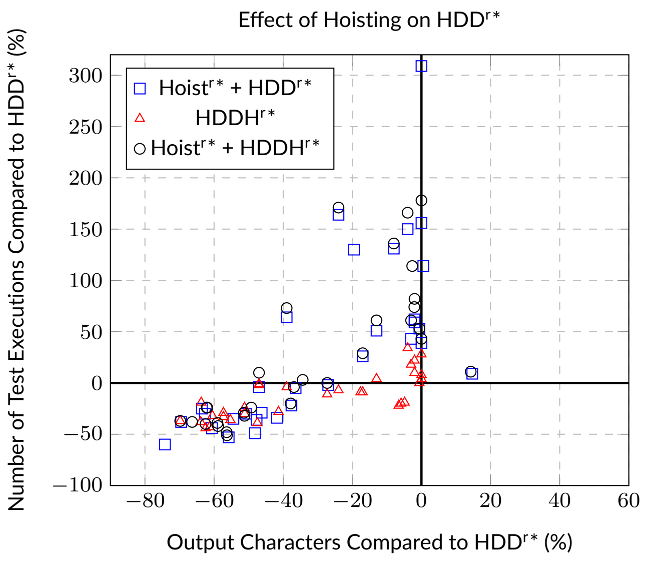
Figure 9 visualizes the raw data from Tables A2, A3, A4, and A5. Each mark on the charts represents a test case reduced with some kind of hoisting applied. The position of the mark reflects how hoisting affected the reduction. Changes in the size of the output are represented along the horizontal axis, while changes in the test executions are represented along the vertical axis (all relative to the size and test step count of the baseline). Marks in the "bottom left" quadrant (a.k.a. quadrant III) are considered the best cases: for the corresponding test cases, hoisting had a positive effect on both the output size and the number of test steps as it has reduced both. Marks in the "top left" and "bottom right" quadrants (a.k.a. quadrants II and IV) represent trade-offs: for those test cases, hoisting either yielded smaller output slower or produced bigger results faster. Marks in the "top right" quadrant (a.k.a. quadrant I) are the cases where hoisting had no benefit at all as both the output size and the number of test steps increased. Figures 9(a) and 9(b) correspond to the results of HDD and HDD$^r$. They show similar patterns: a significant portion of the test cases falls into quadrant III, i.e., they could be reduced faster and further with hoisting. Additionally, another significant portion of the test cases falls into quadrant II, meaning that the test cases could be reduced further with hoisting, although at the cost of more test steps. Figures 9(c) and 9(d) corresponding to the results of Coarse HDD and Coarse HDD$^r$ show different patterns. It is visible that hoisting had no effect on the Coarse HDDH and Coarse HDDH$^r$ algorithm variants. Moreover, the other two algorithm variants (i.e., when hoisting was a preprocessing step) performed exactly the same way. For all algorithm variants and for all combinations, marks are rare in quadrants I and IV. There are only a handful of outliers where hoisting increased the output size.

Table 2 presents aggregated data from both of the used test suites, discussing the influence of the applied hoisting step. Each row has a corresponding row in Table 1, with variant names replaced with values that represent the effect of hoisting compared to the respective baseline, which is in the first column. Table 2a shows output size differences, from which it is clear that hoisting has a positive effect on the reduction outcome – i.e., the final output becomes smaller –, 32.68% improvement with hoisting as a preprocessing step, 14.57% when interlaced with the main reduction algorithm, and 33.65% with the combination of these two approaches. The only exceptions are the Coarse HDDH and Coarse HDDH$^r$ variants that increased the output slightly, compared to the Coarse HDD and Coarse HDD$^r$ respectively.

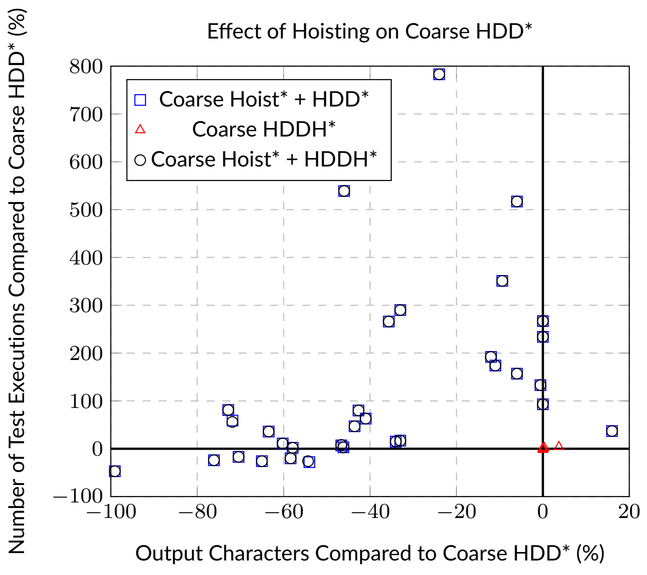
Table 2b shows differences in the number of test executions, i.e., what the effect of hoisting was on the efficiency of the reduction. In general, we have to pay the price for smaller results in the increased number of test steps, which can be quite slow depending on the software under test. When using hoisting as a preprocessing phase, the number of necessary testing steps increased by 69.37%, averaged over all HDDvariants. When using hoisting both as a preprocessing step and interlaced with the HDDvariants, the number of testing steps increased by 72.97%. However, interestingly, using only interlaced hoisting required 4.48% *fewer* tests.



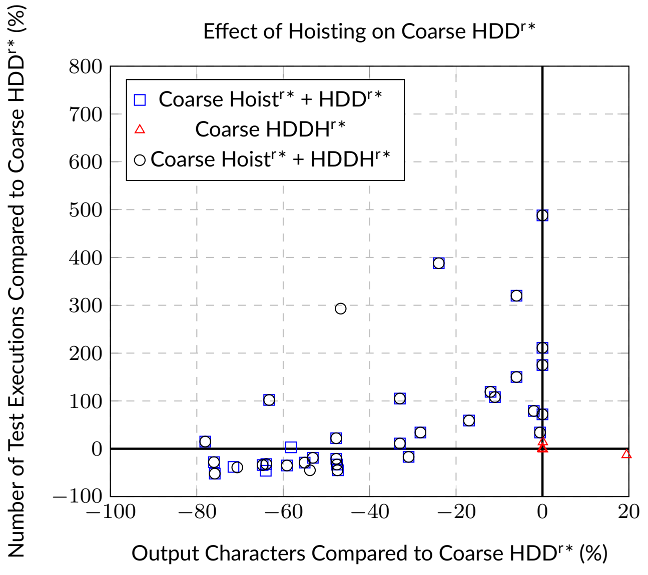
(a)



(b)



(c)



(d)

**FIGURE 9** The effect of hoisting on variants of HDD.

In the above experiments, we have used the number of test executions as our performance metric, which is a common and objective measure. Its alternative, i.e., measuring execution times, could lead to the misinterpretation of the results. E.g., using a faster (or simply a differently configured) machine could have an effect on the measurements and be considered incorrectly as an optimization. However, the reader might still be interested in execution times to assess the practical applicability of the presented techniques. In summary, the time required for reductions ranged from some seconds to several hours in our experiments. Moreover, roughly the same rate of deterioration and improvement could be observed in execution times as in test steps. And we have to note that if we accept that the baseline HDD algorithm and its variants are applicable in practice, then we also have to accept hoisting as practically applicable, since it never changed the order of magnitude of the execution time (or of the test steps) in our experiments. I.e., when the baseline execution time was some seconds then hoisting kept it on the seconds scale, or when the baseline was measured in hours, then the application of hoisting also remained on the hours scale.

Table A1 of Appendix A shows the properties of the inputs used for benchmarking, thus providing opportunities to investigate whether they affect hoisting. It turned out that the height of the parse tree had an effect on the efficiency of hoisting, as shown in Figure 10. The chart is similar to Figure 9, i.e., the vertical axis shows the relative difference compared to the output (Figure 10(a)) or the testing steps (Figure 10(b)) of the baseline

**TABLE 2** Averaged impact of hoisting on different HDD variants

**a** Differences in Output Sizes (%) (Number of Non-whitespace Characters)

| **Algorithm** | **Preprocessing** | **Interlacing** | **Preprocessing & Interlacing** |
|---|---:|---:|---:|
| HDD* | -26.96% | -27.67% | -28.51% |
| HDD$^{r}$* | -29.36% | -31.35% | -32.37% |
| Coarse HDD* | -37.62% | +0.13% | -37.63% |
| Coarse HDD$^{r}$* | -36.77% | +0.62% | -36.07% |

**b** Differences in Number of Test Executions (%)

| **Algorithm** | **Preprocessing** | **Interlacing** | **Preprocessing & Interlacing** |
|---|---:|---:|---:|
| HDD* | +45.54% | -5.28% | +49.59% |
| HDD$^{r}$* | +33.22% | -12.78% | +34.55% |
| Coarse HDD* | +134.15% | +0.13% | +134.13% |
| Coarse HDD$^{r}$* | +64.55% | -0.02% | +73.60% |

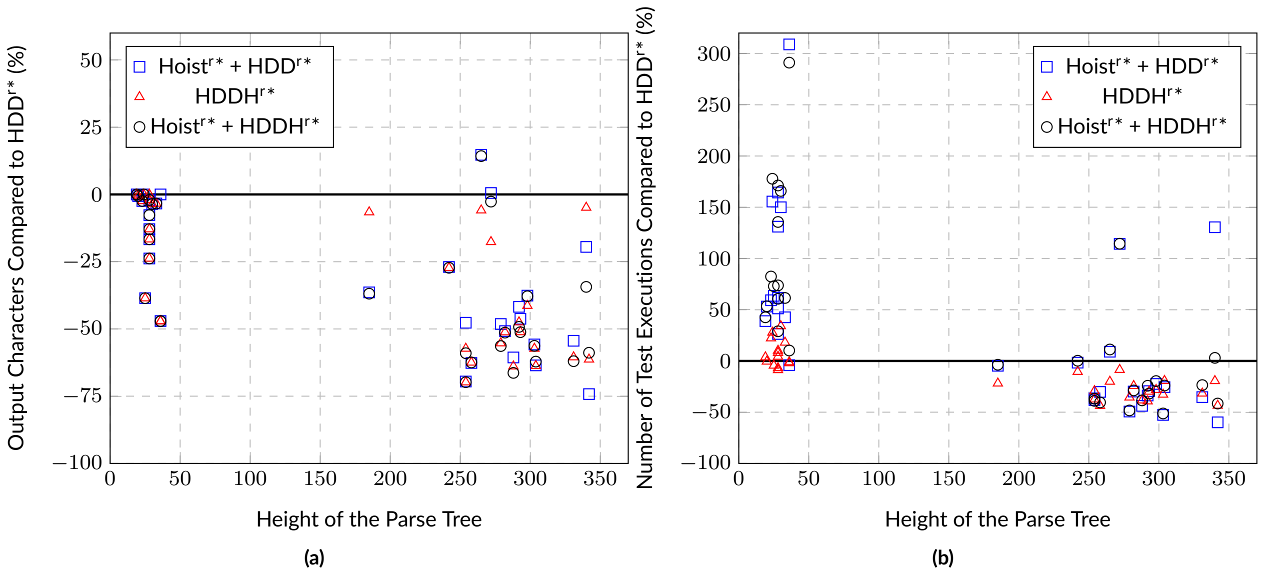


**FIGURE 10** Effect of Hoisting on HDD$^{r}$* as a Function of *Height* of the Parse Tree.

HDD$^{r}$* algorithm. But now the horizontal axis represents the height of the parse tree built from the test cases after squeezing and flattening[7,8]. As discussed above, hoisting had a positive effect on the output size aspect of the reduction in general, however, its effect on the efficiency was inconclusive. While Figure 10(a) confirms the first observation, Figure 10(b) gives an interesting insight into the relation between parse tree height and hoisting efficiency. The chart reveals that if the height of the parse tree is small (below cca. 50), hoisting increases the required testing steps, but if the height of the parse tree is large enough (above 150 in our experiments), hoisting has a mostly positive effect on the number of required testing steps. (Note that the figure contains results from HDD$^{r}$* only, but the same pattern could be observed for the other variants as well.)

## 4.3 | Conclusions

Based on the experimental data and observations above, we can conclude the contributions of this study and answer the research questions:

**Answer to Research Question #1**

There are recurring structures in the parse tree of the preprocessed input that Hierarchical Delta Debugging cannot reduce but a human engineer can easily point out, such as conditional statements, loops, function calls inside a parameter list, etc.

**Answer to Research Question #2**

The Transformation-based Minimization describes an algorithmic framework that enables transformations that can not only remove, but also replace elements in the initial configuration. We have already defined such a transformation – i.e., hoisting – which assumes that a subtree may be replaced by another without losing syntactic correctness if and only if the roots of the subtrees are in an ancestor-descendant relationship.

**Answer to Research Question #3**

- On real-world inputs, hoisting combined with Hierarchical Delta Debugging gives generally smaller, or at least as small outputs as HDD alone. Bigger outputs are rare. Minimized test cases can be as small as ⅕ of the output given by traditional HDD.
- The effects of hoisting to HDD and HDD$^r$ are similar: the majority of the test cases could be reduced further with hoisting.
- Coarse HDD and Coarse HDD$^r$ show similar patterns to the non-coarse variants in respect of the output size: test cases could be reduced further with hoisting. However, hoisting had no effect on the Coarse HDDH and Coarse HDDH$^r$ algorithm variants, furthermore, algorithms performed the reduction exactly the same way when hoisting was a preprocessing step.

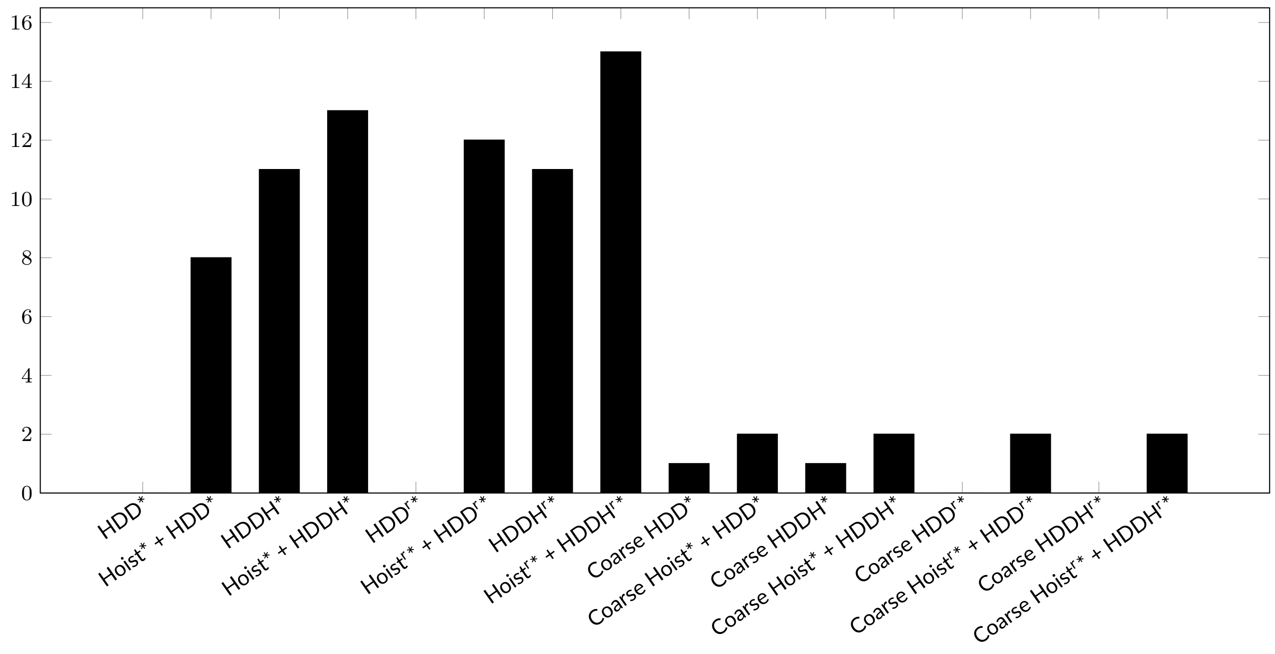
**Answer to Research Question #4**

The effect of hoisting on the efficiency of the reduction highly depends on the height of the input tree. If the height of the tree is small (below 50), hoisting increases the required testing steps. However, if the height of the tree is big enough (above 150), test cases can be reduced faster with hoisting.

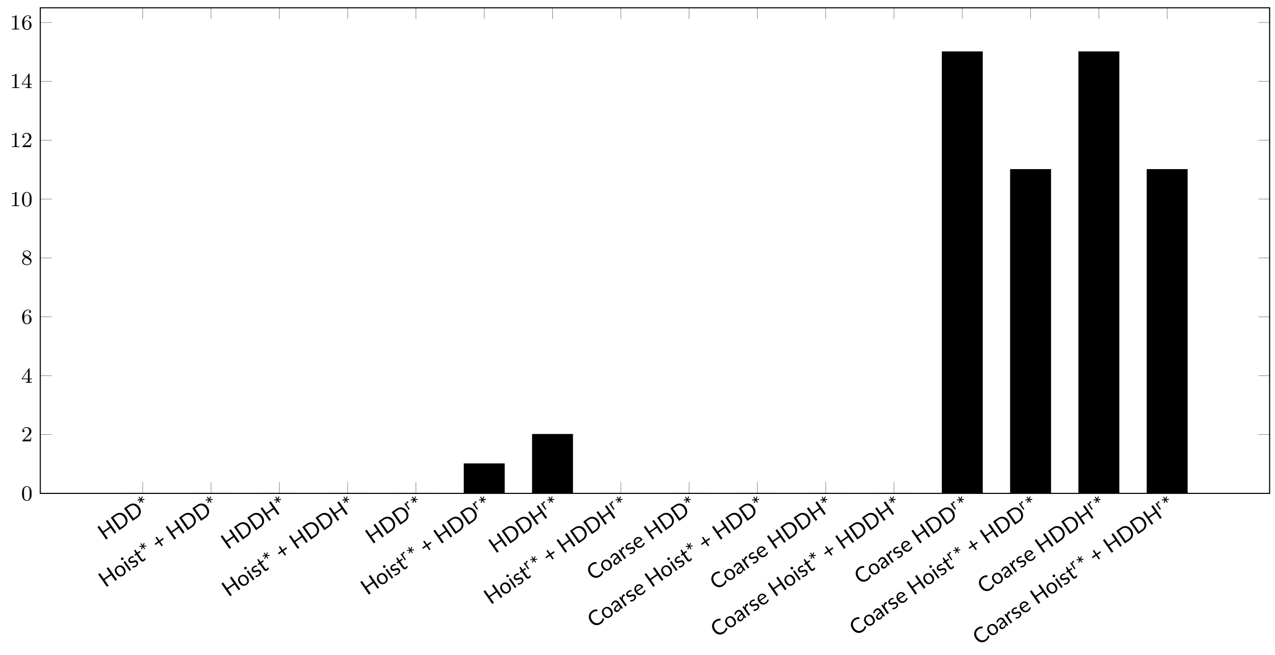
So far, the discussion was only about relative changes compared to the baseline algorithms. However, the reader might be interested in using the best performing variant in an absolute sense and this is what Figure 11 is intended to show. Elements on the horizontal axis correspond to the discussed algorithm variants and the height of the bars above them show how many times they performed best. If multiple algorithm variants produced the same local optimum, all of them were considered the *best*. (Two test cases – namely *jerry-3433* and *jerry-3483* – were excluded from the Figure, since all of the 16 algorithm variants produced exactly the same output.)

Figure 11(a) shows the best-performing algorithm variants based on output characters, i.e., which one produced the smallest output. The first, not so surprising observation is that the Coarse variants are not the most effective variants from the output size point of view. Note that hoisting has a positive effect on the output size (see Table 2a), however, in an absolute manner the Coarse variants fall behind the others effectiveness-wise (see the raw data in Appendix A). The left-hand side of the chart is more promising, especially the Hoist* + HDDH* and Hoist$^{r*}$ + HDDH$^{r*}$ algorithm variants stand out from the rest. Based on these results, if a test must be reduced to the smallest possible size, we would suggest using hoisting in two different phases: first as a preprocessing step and then interlaced with the main algorithm, which could be either HDD or HDD$^r$.

Figure 11(b) shows the same comparison based on the number of test executions, i.e., which one finished the reduction the quickest. The Figure shows the exact opposite view compared to the previous one: the Coarse variants perform the reduction requiring the fewest steps (at the cost of bigger outputs). Based on these observations, if a test must be reduced as quickly as possible, we would suggest using the above formalized Coarse HDD$^{r*}$ algorithm variant. As the experimental results show, hoisting has no effect on this algorithm variant, thus Coarse HDD$^{r*}$ and Coarse HDDH$^{r*}$ work exactly the same way.



**(a)**



**(b)**

**FIGURE 11** Best-performing Algorithms Based on (a) Output Characters, and on (b) Number of Test Executions.

Answer to Research Question #5

- In our experimental results, the Hoist$^{r*}$ + HDDH$^{r*}$ algorithm variant performed the best from an output size point of view, reaching the minimal output in 15 of 30 cases. If the goal is to reduce the inputs as much as possible and the cost of potentially more testing steps is not a big problem, we would suggest using hoisting in two phases: first as a preprocessing step and then interlaced with the HDD$^{r}$ algorithm.

- The newly formalized Coarse HDD$^{r*}$ reduced the most cases (in 15 of 30) using the fewest test attempts, therefore, if the goal is to reduce the inputs as fast as possible and the potentially larger output is not a serious issue, we would suggest using the Coarse HDD$^{r*}$ algorithm variant alone, without hoisting.

### 4.4 | Threats to Validity

Various threats may affect the validity of experimental studies in software engineering. In this work, we identified the following threats and considered the following actions to avoid or minimize such threats:

Selection of test cases: We used two suites of test cases in the format of two programming languages (C and JavaScript). As the formats of the test cases are similar, well-structured, and come from the same domain, we cannot generalize our findings to all types of test cases. However, we think that the results on these test suites are indicative as they contain real-world test cases and have been used in reduction [12,13,14] and fault localization-related studies [15].

Correctness of implementation: To ensure that our experiment implementation is correct and accurate, a code review was conducted by the authors. On selected C and Java examples, we traced the behavior of the implementation to validate that it works as intended. Moreover, the implementation is based on open-source and well-maintained components like the Picireny framework that has been used in several studies [6,7,8,9] and ANTLR v4.

Selection of units of measurement: To avoid bias from formatting differences, we defined size as the number of non-whitespace characters in test cases. Another important aspect of reduction is its performance, or speed. However, measuring the wall clock time of reductions is prone to noise. Therefore, we defined the performance of reduction in the number of test step executions, which can be counted exactly.

Stability of results: We used a dedicated machine for the experiments to avoid noise from unrelated load on the system. Furthermore, we have run our experiments several times to ensure that their results are stable.

## 5 | RELATED WORK

One of the first and most well-known works on automated test case reduction is Delta Debugging by Zeller and Hildebrandt [1,2,3], minimizing inputs of arbitrary format. The price of its generality is a potentially lowered performance because of format-breaking incorrect test cases generated and evaluated during the reduction process. To avoid syntactically broken intermediate test cases, Miserghi and Su proposed to use information about format encoded in context-free grammars, i.e., to convert test cases into a tree representation [4] and apply delta debugging to the levels of the tree. This Hierarchical Delta Debugging approach helped to remove parts of the test case that aligned with syntactic unit boundaries. As a further improvement, Miserghi proposed the concept of syntactically correct replacement for nodes of the tree representation that cannot be completely removed from the test case without causing syntax errors [5].

The formalization of HDD does not detail how to build the tree representation, but its first implementation used traditional context-free grammars to parse the input. To improve on this, Hodován et al. suggested using extended context-free grammars for building the tree [10], thus creating more balanced representations, which could lead to smaller results and improved performance. They have also described various tree transformations with the same goal [8,7].

Tree-based test case reduction does not necessarily mean subtree removal. Bruno suggested using hoisting as an alternative transformation in his framework called SIMP [16], which was specifically designed to reduce database-related inputs. In every reduction step, SIMP tried to replace a node with a compatible descendant. In a follow-up work that introduced the tool FlexMin, Morton and Bruno extended SIMP with Delta Debugging [17]. The main algorithm was the hoisting, while DDMIN was applied only to repeated structures, like lists (column names) and data (string literals). Instead of manually classifying the nodes into two parts, our algorithm tries to hoist every node if it has at least one compatible descendant.

Sun et al. combined the above approaches into their Perses framework [12]. In their work, they utilized quantifiers and normalized the parse tree producing grammars by rewriting recursive rules to use quantified expressions. This transformed grammar form was referred to as Perses Normal Form (PNF). During the reduction, they applied a worklist algorithm, in which non-terminals with more tokens were prioritized over nodes with fewer token descendants. In every step, a node was popped from the worklist and reduced according to its type: quantified nodes were reduced with DDMIN while hoisting was applied on non-quantified, regular ones. They also mentioned that the number of compatible nodes for hoisting can be enormous. Instead of collecting all candidates and trying to hoist the descendant that has the longest distance from its ancestor, they limited the search space with two constraints: (1) the number of nodes between ancestor-descendants is limited (4 was used in their evaluation); (2) if a compatible descendant has been found, the searching was not continued further down in the hierarchy.

Built upon the ideas introduced in Perses, Gharachorlu and Sumner [13] extended it in a new framework, named Pardis, with an improved queue prioritization algorithm. One of the key differences was that eventually, they eliminated the hoisting step, since they found it too expensive from a performance perspective.

Herfert et al. [18] also combined subtree removal and hoisting in their Generalized Tree Reduction (GTR) algorithm but instead of analyzing a grammar to decide about the applicability of a certain transformation they learned this information from an existing test corpus. The search-based program repair work of Gazzola et al. [19] also mentions modifications on the abstract syntax tree, however, transformations are given as predefined templates and could be performed randomly without the ancestor-descendant relationship.

The above-mentioned works targeted textual inputs, but test case reduction can be applied to other scenarios as well. Several authors have minimized faulty event sequences originating from various sources: Scott et al.[20] minimized event sequences of distributed systems, Clapp et al.[21] aimed at Android GUI event sequences with a variant of DDMIN. Moreover, Delta Debugging was even used for the minimization of SMT solver formulas (Brummayer et al.[22]).

An interesting analogy between test case reduction and program slicing was recognized by Binkley et al[23,24,25,26,27,28]. They have realized that the concepts of slicing (e.g., the program to be sliced or the slicing criterion) can be reformulated as concepts of test case reduction (e.g., the test case or the interestingness property, respectively). Their approach, called observation-based slicing, avoids the complexities of building a dependency graph representation of a program and can work purely at the syntactic level. Although their approach is not DDMIN-based, the algorithms show similarities with the ideas of DDMIN and HDD.

We can consider the algorithms as phases in our implementation that process the input via tree traversal. The pruning phase is identical to HDD, while Hoist phase tries to hoist the furthest compatible descendant from the tree hierarchy. In addition, other transformations can be implemented easily as new phases into the framework.

## 6 | SUMMARY

In this paper, we have been focusing on the automated reduction of tree-structured test cases. We have investigated the structure of the input of the well-known Hierarchical Delta Debugging and found that there are recurring structures in the input tree that HDD cannot reduce, however, human engineers easily can. Motivated by this discovery, we described an algorithmic framework, the Transformation-based Minimization that enables transformations on the input beyond the well-established pruning. Using this framework, we proposed an extension to the pruning-based Hierarchical Delta Debugging technique, called hoisting. The key idea of the extension is to allow moving up subtrees in the structure of the test case without losing syntactic correctness, thus reducing its size. As several optimizations have been published to the original HDD since its invention, we have investigated the effect of our extension on multiple variants: HDD, HDD$^r$, Coarse HDD, and Coarse HDD$^r$ (the latter of which is first defined in this paper). We have also described various ways of how hoisting can be combined with the different HDD-based reduction algorithms.

We prototyped the introduced algorithms and evaluated several combinations of hoisting and pruning on multiple test suites. The results of our experiments support that hoisting can improve the usability of the output of HDD-based algorithms by producing as much as 80% smaller minimized tests in the best case. In general, any kind of hoisting-pruning combination has a positive effect on the reduction outcome: 32.68% smaller outputs on average with hoisting as a preprocessing step, 14.57% on average as interlaced with the main reduction algorithm, and 33.65% on average with the combination of these two approaches. Only the Coarse variants show a different behavior: interlacing hoisting with Coarse HDD or Coarse HDD$^r$ increased the size of the output slightly, compared to the Coarse HDD and Coarse HDD$^r$, respectively.

Our primary goal was to shrink the output of the reduction even further, however, the study is not complete without presenting the effect of hoisting on the efficiency of the reduction process. In general, our experimental results show that we have to pay the price of smaller results in the increased number of testing steps. Using hoisting as a preprocessing phase increased the testing steps by 69.37% on average, while the combination of the preprocessing and interlaced hoisting approaches gives 72.97% more testing steps on average. However, simply interlacing hoisting with the main algorithm required 4.48% fewer calls. The analysis of the internal structure of the inputs shows that hoisting has a positive effect on the efficiency of the reduction if the height of the input tree is large enough, otherwise hoisting increases the number of required testing steps.

In our experiments, the Hoist$^{r*}$ + HDDH$^{r*}$ algorithm variant outperformed the others, producing the smallest output in 50% of the benchmark cases. If the goal of an engineer is to reduce the inputs as much as possible and the cost of potentially more testing steps is not considered to be an issue, we suggest using hoisting in two phases: first as a preprocessing step and then interlaced with the HDD$^r$ algorithm.

As for future work, we have plans to continue this topic of research in various ways. We wish to conduct further experiments to ensure that the results generalize to inputs that are larger or differently structured compared to those investigated in this paper. We also aim at speeding up hoisting-extended HDD while not increasing the output size. Furthermore, we would like to explore additional minimizing transformations. Finally, we are also interested in the human understandability aspect of reduced test cases, considering both pruning- and transformation-based reduction methods.

## Acknowledgment

This research was supported by project TKP2021-NVA-09, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

## Data Availability Statement

The source code of the implemented algorithms are openly available in the Picireny project at https://github.com/renatahodovan/picireny. Input data to the implemented algorithms are openly available in the JerryScript Reduction Test Suite project at https://github.com/vincedani/jrts, and in the Perses project at https://github.com/choderalab/perses/. Raw execution log data of the implemented algorithms are available from the corresponding author upon request.

## References

1. Zeller A. Yesterday, My Program Worked. Today, It Does Not. Why?. In: *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly With 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (ESEC/FSE)*. 1687 of *Lecture Notes in Comput. Sci.* Springer; 1999: 253–267.
2. Hildebrandt R, Zeller A. Simplifying Failure-Inducing Input. In: *Proc. 2000 ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*ACM; 2000: 135–145.
3. Zeller A, Hildebrandt R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 2002; 28(2): 183–200.
4. Misherghi G, Su Z. HDD: Hierarchical Delta Debugging. In: *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*ACM; 2006: 142–151.
5. Misherghi GS. Hierarchical Delta Debugging. Master's thesis. Univ. California, Davis. California: 2007.
6. Hodován R, Kiss Á. Practical Improvements to the Minimizing Delta Debugging Algorithm. In: *Proc. 11th Int. Joint Conf. Softw. Technol. (ICSOFT)*. 1. SciTePress; 2016: 241–248
7. Hodován R, Kiss Á, Gyimóthy T. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. In: *Proc. 12th IEEE/ACM Int. Workshop Automat. Softw. Test. (AST)*IEEE; 2017: 23–29
8. Hodován R, Kiss Á, Gyimóthy T. Coarse Hierarchical Delta Debugging. In: *Proc. 33rd IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*IEEE; 2017: 194–203
9. Kiss Á, Hodován R, Gyimóthy T. HDDr: A Recursive Variant of the Hierarchical Delta Debugging Algorithm. In: *Proc. 9th ACM SIGSOFT Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*ACM; 2018: 16–22
10. Hodován R, Kiss Á. Modernizing Hierarchical Delta Debugging. In: *Proc. 7th Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*ACM; 2016: 31–37
11. Vince D, Hodován R, Bársony D, Kiss Á. Extending Hierarchical Delta Debugging with Hoisting. In: *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*; 2021: 60-69
12. Sun C, Li Y, Zhang Q, Gu T, Su Z. Perses: Syntax-Guided Program Reduction. In: *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*ACM; 2018: 361–371.
13. Gharachorlu G, Sumner N. PARDIS: Priority Aware Test Case Reduction. In: *Proc. 22nd Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*. 11424 of *Lecture Notes in Comput. Sci.* Springer; 2019: 409–426.
14. Kiss Á. Generalizing the Split Factor of the Minimizing Delta Debugging Algorithm. *IEEE Access* 2020; 8: 219837–219846. doi: 10.1109/ACCESS.2020.3043027
15. Vince D, Hodován R, Kiss Á. Reduction-assisted Fault Localization: Don't Throw Away the By-products!. In: *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*SciTePress; 2021: 196–206
16. Bruno N. Minimizing Database Repros Using Language Grammars. In: *Proc. 13th Int. Conf. Extending Database Technol. (EDBT)*ACM; 2010: 382–393.
17. Morton K, Bruno N. FlexMin: A Flexible Tool for Automatic Bug Isolation in DBMS Software. In: *Proc. 4th Int. Workshop Test. Database Syst. (DBTest)*ACM; 2011: 1:1–1:6.
18. Herfert S, Patra J, Pradel M. Automatically Reducing Tree-Structured Test Inputs. In: *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*IEEE; 2017: 861–871.

19. Gazzola L, Micucci D, Mariani L. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* 2019; 45(1): 34–67. doi: 10.1109/TSE.2017.2755013

20. Scott C, Brajkovic V, Necula G, Krishnamurthy A, Shenker S. Minimizing Faulty Executions of Distributed Systems. In: *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*USENIX Association; 2016: 291–309.

21. Clapp L, Bastani O, Anand S, Aiken A. Minimizing GUI Event Traces. In: *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*ACM; 2016: 422–434.

22. Brummayer R, Biere A. Fuzzing and Delta-debugging SMT Solvers. In: *Proc. 7th Int. Workshop Satisfiability Modulo Theories (SMT)*ACM; 2009: 1–5.

23. Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S. ORBS: Language-Independent Program Slicing. In: *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*ACM; 2014: 109–120.

24. Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S. ORBS and the limits of static slicing. In: *Proc. 15th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*IEEE; 2015: 1–10.

25. Yoo S, Binkley D, Eastman R. Seeing Is Slicing: Observation Based Slicing of Picture Description Languages. In: *Proc. 14th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*IEEE; 2014: 175–184.

26. Gold NE, Binkley D, Harman M, Islam S, Krinke J, Yoo S. Generalized Observational Slicing for Tree-Represented Modelling Languages. In: *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*ACM; 2017: 547–558.

27. Binkley D, Gold N, Islam S, Krinke J, Yoo S. Tree-Oriented vs. Line-Oriented Observation-Based Slicing. In: *Proc. 17th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*IEEE; 2017: 21–30.

28. Binkley D, Gold N, Islam S, Krinke J, Yoo S. A comparison of tree- and line-oriented observational slicing. *Empirical Softw. Eng.* 2019; 24(5): 3077–3113. doi: 10.1007/s10664-018-9675-9

## APPENDIX

## A EXPERIMENT DATA

In the tables below, size is expressed as the number of non-whitespace characters to avoid bias from indentation or other formatting differences. In each row of all tables, bold numbers highlight the best result(s).

**TABLE A1** Properties of the Inputs Used for Benchmarking

| Test | Size | Tree Height | Rules | Tokens |
|---|---:|---:|---:|---:|
| clang-22382 | 65,786 | 242 | 29,344 | 6,573 |
| clang-22704 | 597,827 | 272 | 255,972 | 61,255 |
| clang-23309 | 118,178 | 288 | 52,183 | 11,570 |
| clang-23353 | 94,734 | 185 | 44,100 | 9,989 |
| clang-25900 | 245,065 | 292 | 106,751 | 23,406 |
| clang-26350 | 378,160 | 304 | 168,324 | 25,790 |
| clang-26760 | 588,548 | 340 | 288,964 | 60,762 |
| clang-27747 | 409,083 | 265 | 238,604 | 46,295 |
| clang-31259 | 137,161 | 331 | 66,291 | 14,590 |
| gcc-59903 | 166,754 | 298 | 76,531 | 17,322 |
| gcc-60116 | 218,223 | 279 | 100,651 | 21,479 |
| gcc-61383 | 110,643 | 303 | 46,786 | 9,070 |
| gcc-61917 | 254,742 | 254 | 115,834 | 24,508 |
| gcc-64990 | 439,587 | 342 | 200,107 | 45,000 |
| gcc-65383 | 125,221 | 254 | 58,846 | 13,237 |
| gcc-66186 | 139,087 | 258 | 65,228 | 14,434 |
| gcc-66375 | 191,827 | 282 | 86,512 | 19,216 |
| gcc-70127 | 400,556 | 293 | 210,039 | 44,942 |
| gcc-71626 | 14,465 | 20 | 8,044 | 2,047 |
| jerry-3299 | 1,208 | 33 | 608 | 140 |
| jerry-3361 | 1,520 | 28 | 562 | 163 |
| jerry-3376 | 4,647 | 36 | 2,194 | 473 |
| jerry-3408 | 2,100 | 28 | 778 | 228 |
| jerry-3431 | 648 | 30 | 527 | 130 |
| jerry-3433 | 652 | 24 | 378 | 82 |
| jerry-3437 | 4,623 | 36 | 2,188 | 471 |
| jerry-3479 | 3,998 | 25 | 1,326 | 347 |
| jerry-3483 | 326 | 19 | 193 | 48 |
| jerry-3506 | 2,735 | 28 | 1,278 | 343 |
| jerry-3523 | 2,802 | 28 | 1,416 | 345 |
| jerry-3534 | 1,409 | 28 | 641 | 176 |
| jerry-3536 | 592 | 23 | 310 | 71 |

**TABLE A2** Hoisting and HDD

**a** Output Sizes (Number of Non-whitespace Characters)

| Test | HDD$^*$ | Hoist$^*$+HDD$^*$ | HDDH$^*$ | Hoist$^*$+HDDH$^*$ |
|---|---|---|---|---|
| clang-22382 | 582 | 489 *(-15.98%)* | **475** *(-18.38%)* | 489 *(-15.98%)* |
| clang-22704 | 168 | 164 *(-2.38%)* | 165 *(-1.79%)* | **161** *(-4.17%)* |
| clang-23309 | 3,582 | 1,486 *(-58.51%)* | 1,677 *(-53.18%)* | **1,416** *(-60.47%)* |
| clang-23353 | 374 | 354 *(-5.35%)* | 592 *(+58.29%)* | **351** *(-6.15%)* |
| clang-25900 | 1,562 | 986 *(-36.88%)* | **885** *(-43.34%)* | 888 *(-43.15%)* |
| clang-26350 | 1,613 | 778 *(-51.77%)* | **585** *(-63.73%)* | 760 *(-52.88%)* |
| clang-26760 | 586 | 595 *(+1.54%)* | **297** *(-49.32%)* | 582 *(-0.68%)* |
| clang-27747 | 419 | 406 *(-3.10%)* | **377** *(-10.02%)* | 415 *(-0.95%)* |
| clang-31259 | 2,174 | 814 *(-62.56%)* | 947 *(-56.44%)* | **796** *(-63.39%)* |
| gcc-59903 | 1,726 | 1,432 *(-17.03%)* | **620** *(-64.08%)* | 1,298 *(-24.80%)* |
| gcc-60116 | 3,788 | 1,185 *(-68.72%)* | 1,152 *(-69.59%)* | **941** *(-75.16%)* |
| gcc-61383 | 1,701 | 1,041 *(-38.80%)* | **844** *(-50.38%)* | 874 *(-48.62%)* |
| gcc-61917 | 1,764 | 575 *(-67.40%)* | 885 *(-49.83%)* | **570** *(-67.69%)* |
| gcc-64990 | 2,844 | 561 *(-80.27%)* | 1,282 *(-54.92%)* | **551** *(-80.63%)* |
| gcc-65383 | 1,027 | 543 *(-47.13%)* | 490 *(-52.29%)* | **441** *(-57.06%)* |
| gcc-66186 | 2,614 | 978 *(-62.59%)* | **977** *(-62.62%)* | **977** *(-62.62%)* |
| gcc-66375 | 2,963 | 1,446 *(-51.20%)* | 1,439 *(-51.43%)* | **1,430** *(-51.74%)* |
| gcc-70127 | – | 992 – | **915** – | 947 – |
| gcc-71626 | 168 | **167** *(-0.60%)* | **167** *(-0.60%)* | **167** *(-0.60%)* |
| jerry-3299 | 92 | **89** *(-3.26%)* | **89** *(-3.26%)* | **89** *(-3.26%)* |
| jerry-3361 | 97 | **95** *(-2.06%)* | **95** *(-2.06%)* | **95** *(-2.06%)* |
| jerry-3376 | 70 | **35** *(-50.00%)* | 37 *(-47.14%)* | **35** *(-50.00%)* |
| jerry-3408 | 62 | **54** *(-12.90%)* | **54** *(-12.90%)* | **54** *(-12.90%)* |
| jerry-3431 | 28 | **27** *(-3.57%)* | **27** *(-3.57%)* | **27** *(-3.57%)* |
| jerry-3433 | **18** | **18** – | **18** – | **18** – |
| jerry-3437 | 34 | **18** *(-47.06%)* | **18** *(-47.06%)* | **18** *(-47.06%)* |
| jerry-3479 | 94 | **89** *(-5.32%)* | **89** *(-5.32%)* | **89** *(-5.32%)* |
| jerry-3483 | **38** | **38** – | **38** – | **38** – |
| jerry-3506 | **52** | **52** – | **52** – | **52** – |
| jerry-3523 | 63 | **48** *(-23.81%)* | **48** *(-23.81%)* | **48** *(-23.81%)* |
| jerry-3534 | 96 | **80** *(-16.67%)* | **80** *(-16.67%)* | **80** *(-16.67%)* |
| jerry-3536 | 123 | **120** *(-2.44%)* | **120** *(-2.44%)* | **120** *(-2.44%)* |

**b** Number of Test Executions

| Test | HDD$^*$ | Hoist$^*$+HDD$^*$ | HDDH$^*$ | Hoist$^*$+HDDH$^*$ |
|---|---|---|---|---|
| clang-22382 | 14,699 | **9,910** *(-32.58%)* | 12,955 *(-11.86%)* | 9,997 *(-31.99%)* |
| clang-22704 | 10,540 | 21,094 *(+100.13%)* | **10,474** *(-0.63%)* | 21,180 *(+100.95%)* |
| clang-23309 | 24,630 | 16,025 *(-34.94%)* | 19,833 *(-19.48%)* | **15,828** *(-35.74%)* |
| clang-23353 | **14,598** | 30,114 *(+106.29%)* | 14,662 *(+0.44%)* | 30,182 *(+106.75%)* |
| clang-25900 | 14,766 | 9,983 *(-32.39%)* | 12,510 *(-15.28%)* | **9,865** *(-33.19%)* |
| clang-26350 | 16,789 | 18,851 *(+12.28%)* | **14,831** *(-11.66%)* | 19,847 *(+18.21%)* |
| clang-26760 | 12,957 | **11,808** *(-8.87%)* | 11,884 *(-8.28%)* | 11,835 *(-8.66%)* |
| clang-27747 | 7,174 | 13,899 *(+93.74%)* | **6,601** *(-7.99%)* | 13,911 *(+93.91%)* |
| clang-31259 | 19,239 | **8,791** *(-54.31%)* | 15,914 *(-17.28%)* | 8,992 *(-53.26%)* |
| gcc-59903 | 18,935 | 12,554 *(-33.70%)* | 18,381 *(-2.93%)* | **12,345** *(-34.80%)* |
| gcc-60116 | 23,844 | 12,740 *(-46.57%)* | 17,153 *(-28.06%)* | **12,041** *(-49.50%)* |
| gcc-61383 | 17,286 | **11,802** *(-31.73%)* | 15,350 *(-11.20%)* | 11,984 *(-30.67%)* |
| gcc-61917 | 17,455 | **8,432** *(-51.69%)* | 13,769 *(-21.12%)* | 8,525 *(-51.16%)* |
| gcc-64990 | 19,624 | **10,533** *(-46.33%)* | 17,565 *(-10.49%)* | 10,548 *(-46.25%)* |
| gcc-65383 | 16,239 | 6,524 *(-59.83%)* | 11,801 *(-27.33%)* | **6,334** *(-61.00%)* |
| gcc-66186 | 16,181 | 13,771 *(-14.89%)* | 13,930 *(-13.91%)* | **13,762** *(-14.95%)* |
| gcc-66375 | 21,251 | **16,046** *(-24.49%)* | 18,393 *(-13.45%)* | 16,131 *(-24.09%)* |
| gcc-70127 | – | **15,699** – | 18,330 – | 15,974 – |
| gcc-71626 | 4,216 | 6,520 *(+54.65%)* | **4,205** *(-0.26%)* | 6,522 *(+54.70%)* |
| jerry-3299 | **176** | 228 *(+29.55%)* | 192 *(+9.09%)* | 251 *(+42.61%)* |
| jerry-3361 | **144** | 254 *(+76.39%)* | 154 *(+6.94%)* | 266 *(+84.72%)* |
| jerry-3376 | 119 | 412 *(+246.22%)* | **109** *(-8.40%)* | 422 *(+254.62%)* |
| jerry-3408 | **167** | 278 *(+66.47%)* | 178 *(+6.59%)* | 289 *(+73.05%)* |
| jerry-3431 | **55** | 185 *(+236.36%)* | 70 *(+27.27%)* | 192 *(+249.09%)* |
| jerry-3433 | **18** | 58 *(+222.22%)* | 23 *(+27.78%)* | 62 *(+244.44%)* |
| jerry-3437 | 49 | 49 – | **48** *(-2.04%)* | 56 *(+14.29%)* |
| jerry-3479 | 233 | 576 *(+147.21%)* | **230** *(-1.29%)* | 592 *(+154.08%)* |
| jerry-3483 | **69** | 95 *(+37.68%)* | 71 *(+2.90%)* | 97 *(+40.58%)* |
| jerry-3506 | **115** | 248 *(+115.65%)* | 122 *(+6.09%)* | 251 *(+118.26%)* |
| jerry-3523 | 111 | 416 *(+274.77%)* | **83** *(-25.23%)* | 421 *(+279.28%)* |
| jerry-3534 | 173 | 197 *(+13.87%)* | **149** *(-13.87%)* | 200 *(+15.61%)* |
| jerry-3536 | **150** | 226 *(+50.67%)* | 182 *(+21.33%)* | 251 *(+67.33%)* |

**TABLE A3** Hoisting and $\text{HDD}^{r}$

**a** Output Sizes (Number of Non-whitespace Characters)

| Test | $\text{HDD}^{r*}$ | $\text{Hoist}^{r*}$+$\text{HDD}^{r*}$ | | $\text{HDDH}^{r*}$ | | $\text{Hoist}^{r*}$+$\text{HDDH}^{r*}$ | |
|---|---|---|---|---|---|---|---|
| clang-22382 | 671 | 490 | *(-26.97%)* | **488** | *(-27.27%)* | **488** | *(-27.27%)* |
| clang-22704 | 187 | 188 | *(+0.53%)* | **154** | *(-17.65%)* | 182 | *(-2.67%)* |
| clang-23309 | 4,338 | 1,709 | *(-60.60%)* | 1,567 | *(-63.88%)* | **1,460** | *(-66.34%)* |
| clang-23353 | 607 | 386 | *(-36.41%)* | 567 | *(-6.59%)* | **383** | *(-36.90%)* |
| clang-25900 | 1,651 | 961 | *(-41.79%)* | 867 | *(-47.49%)* | **838** | *(-49.24%)* |
| clang-26350 | 1,651 | 602 | *(-63.54%)* | **599** | *(-63.72%)* | 625 | *(-62.14%)* |
| clang-26760 | 451 | 363 | *(-19.51%)* | 429 | *(-4.88%)* | **296** | *(-34.37%)* |
| clang-27747 | 442 | 507 | *(+14.71%)* | **416** | *(-5.88%)* | 505 | *(+14.25%)* |
| clang-31259 | 2,136 | 974 | *(-54.40%)* | 844 | *(-60.49%)* | **812** | *(-61.99%)* |
| gcc-59903 | 2,536 | 1,582 | *(-37.62%)* | **1,487** | *(-41.36%)* | 1,576 | *(-37.85%)* |
| gcc-60116 | 3,355 | 1,739 | *(-48.17%)* | 1,500 | *(-55.29%)* | **1,464** | *(-56.36%)* |
| gcc-61383 | 1,569 | 694 | *(-55.77%)* | **670** | *(-57.30%)* | 686 | *(-56.28%)* |
| gcc-61917 | 1,904 | 580 | *(-69.54%)* | **575** | *(-69.80%)* | **575** | *(-69.80%)* |
| gcc-64990 | 1,497 | **386** | *(-74.22%)* | 579 | *(-61.32%)* | 616 | *(-58.85%)* |
| gcc-65383 | 1,042 | 545 | *(-47.70%)* | 445 | *(-57.29%)* | **427** | *(-59.02%)* |
| gcc-66186 | 2,592 | **968** | *(-62.65%)* | 973 | *(-62.46%)* | 973 | *(-62.46%)* |
| gcc-66375 | 2,805 | 1,380 | *(-50.80%)* | 1,369 | *(-51.19%)* | **1,364** | *(-51.37%)* |
| gcc-70127 | 1,830 | 984 | *(-46.23%)* | 894 | *(-51.15%)* | **893** | *(-51.20%)* |
| gcc-71626 | 168 | **167** | *(-0.60%)* | **167** | *(-0.60%)* | **167** | *(-0.60%)* |
| jerry-3299 | 89 | **86** | *(-3.37%)* | **86** | *(-3.37%)* | **86** | *(-3.37%)* |
| jerry-3361 | 97 | **95** | *(-2.06%)* | **95** | *(-2.06%)* | **95** | *(-2.06%)* |
| jerry-3376 | 70 | 70 | – | **37** | *(-47.14%)* | **37** | *(-47.14%)* |
| jerry-3408 | 62 | **54** | *(-12.90%)* | **54** | *(-12.90%)* | **54** | *(-12.90%)* |
| jerry-3431 | 28 | **27** | *(-3.57%)* | **27** | *(-3.57%)* | **27** | *(-3.57%)* |
| jerry-3433 | **18** | **18** | – | **18** | – | **18** | – |
| jerry-3437 | 34 | **18** | *(-47.06%)* | **18** | *(-47.06%)* | **18** | *(-47.06%)* |
| jerry-3479 | 140 | **86** | *(-38.57%)* | **86** | *(-38.57%)* | **86** | *(-38.57%)* |
| jerry-3483 | **38** | **38** | – | **38** | – | **38** | – |
| jerry-3506 | 52 | **48** | *(-7.69%)* | 52 | – | **48** | *(-7.69%)* |
| jerry-3523 | 63 | **48** | *(-23.81%)* | **48** | *(-23.81%)* | **48** | *(-23.81%)* |
| jerry-3534 | 96 | **80** | *(-16.67%)* | **80** | *(-16.67%)* | **80** | *(-16.67%)* |
| jerry-3536 | 123 | **120** | *(-2.44%)* | **120** | *(-2.44%)* | **120** | *(-2.44%)* |

**b** Number of Test Executions

| Test | $\text{HDD}^{r*}$ | $\text{Hoist}^{r*}$+$\text{HDD}^{r*}$ | | $\text{HDDH}^{r*}$ | | $\text{Hoist}^{r*}$+$\text{HDDH}^{r*}$ | |
|---|---|---|---|---|---|---|---|
| clang-22382 | 3,522 | 3,464 | *(-1.65%)* | **3,152** | *(-10.51%)* | 3,530 | *(+0.23%)* |
| clang-22704 | 2,917 | 6,249 | *(+114.23%)* | **2,667** | *(-8.57%)* | 6,254 | *(+114.40%)* |
| clang-23309 | 10,896 | **6,121** | *(-43.82%)* | 6,716 | *(-38.36%)* | 6,705 | *(-38.46%)* |
| clang-23353 | 5,087 | 4,848 | *(-4.70%)* | **3,976** | *(-21.84%)* | 4,900 | *(-3.68%)* |
| clang-25900 | 5,196 | 3,446 | *(-33.68%)* | **3,155** | *(-39.28%)* | 3,939 | *(-24.19%)* |
| clang-26350 | 9,071 | **6,759** | *(-25.49%)* | 7,319 | *(-19.31%)* | 6,897 | *(-23.97%)* |
| clang-26760 | 4,254 | 9,805 | *(+130.49%)* | **3,426** | *(-19.46%)* | 4,382 | *(+3.01%)* |
| clang-27747 | 3,321 | 3,621 | *(+9.03%)* | **2,653** | *(-20.11%)* | 3,693 | *(+11.20%)* |
| clang-31259 | 5,654 | **3,660** | *(-35.27%)* | 3,864 | *(-31.66%)* | 4,323 | *(-23.54%)* |
| gcc-59903 | 8,302 | 6,439 | *(-22.44%)* | **5,966** | *(-28.14%)* | 6,674 | *(-19.61%)* |
| gcc-60116 | 11,371 | **5,759** | *(-49.35%)* | 7,332 | *(-35.52%)* | 5,881 | *(-48.28%)* |
| gcc-61383 | 6,229 | **2,949** | *(-52.66%)* | 4,190 | *(-32.73%)* | 3,038 | *(-51.23%)* |
| gcc-61917 | 5,935 | **3,668** | *(-38.20%)* | 3,715 | *(-37.41%)* | 3,754 | *(-36.75%)* |
| gcc-64990 | 5,446 | **2,185** | *(-59.88%)* | 3,077 | *(-43.50%)* | 3,184 | *(-41.54%)* |
| gcc-65383 | 4,126 | 2,622 | *(-36.45%)* | 2,919 | *(-29.25%)* | **2,520** | *(-38.92%)* |
| gcc-66186 | 5,281 | 3,685 | *(-30.22%)* | **2,969** | *(-43.78%)* | 3,143 | *(-40.48%)* |
| gcc-66375 | 5,613 | **3,938** | *(-29.84%)* | 4,251 | *(-24.27%)* | 3,999 | *(-28.75%)* |
| gcc-70127 | 5,728 | 4,043 | *(-29.42%)* | 4,019 | *(-29.84%)* | **3,906** | *(-31.81%)* |
| gcc-71626 | 620 | 949 | *(+53.06%)* | **617** | *(-0.48%)* | 951 | *(+53.39%)* |
| jerry-3299 | **122** | 174 | *(+42.62%)* | 144 | *(+18.03%)* | 197 | *(+61.48%)* |
| jerry-3361 | **99** | 160 | *(+61.62%)* | 109 | *(+10.10%)* | 172 | *(+73.74%)* |
| jerry-3376 | 90 | 368 | *(+308.89%)* | **89** | *(-1.11%)* | 352 | *(+291.11%)* |
| jerry-3408 | **117** | 177 | *(+51.28%)* | 122 | *(+4.27%)* | 188 | *(+60.68%)* |
| jerry-3431 | **44** | 110 | *(+150.00%)* | 59 | *(+34.09%)* | 117 | *(+165.91%)* |
| jerry-3433 | **18** | 46 | *(+155.56%)* | 23 | *(+27.78%)* | 50 | *(+177.78%)* |
| jerry-3437 | 49 | **47** | *(-4.08%)* | 48 | *(-2.04%)* | 54 | *(+10.20%)* |
| jerry-3479 | 181 | 296 | *(+63.54%)* | **173** | *(-4.42%)* | 313 | *(+72.93%)* |
| jerry-3483 | **54** | 75 | *(+38.89%)* | 56 | *(+3.70%)* | 77 | *(+42.59%)* |
| jerry-3506 | **87** | 201 | *(+131.03%)* | 94 | *(+8.05%)* | 205 | *(+135.63%)* |
| jerry-3523 | 73 | 193 | *(+164.38%)* | **68** | *(-6.85%)* | 198 | *(+171.23%)* |
| jerry-3534 | 114 | 144 | *(+26.32%)* | **104** | *(-8.77%)* | 147 | *(+28.95%)* |
| jerry-3536 | **108** | 172 | *(+59.26%)* | 132 | *(+22.22%)* | 197 | *(+82.41%)* |

**TABLE A4** Coarse Hoisting and HDD

**a** Output Sizes (Number of Non-whitespace Characters)

| Test | Coarse HDD* | Coarse Hoist*+HDD* | | Coarse HDDH* | | Coarse Hoist*+HDDH* | |
|---|---|---|---|---|---|---|---|
| clang-22382 | 975 | **643** | *(-34.05%)* | 975 | – | **643** | *(-34.05%)* |
| clang-22704 | 351 | **318** | *(-9.40%)* | 351 | – | **318** | *(-9.40%)* |
| clang-23309 | 5,556 | **2,312** | *(-58.39%)* | 5,556 | – | **2,312** | *(-58.39%)* |
| clang-23353 | 49,114 | **457** | *(-99.07%)* | 49,114 | – | **457** | *(-99.07%)* |
| clang-25900 | 2,472 | **1,331** | *(-46.16%)* | 2,472 | – | **1,331** | *(-46.16%)* |
| clang-26350 | 2,936 | **798** | *(-72.82%)* | 2,936 | – | **798** | *(-72.82%)* |
| clang-26760 | 1,235 | **708** | *(-42.67%)* | 1,235 | – | **708** | *(-42.67%)* |
| clang-27747 | 973 | **626** | *(-35.66%)* | 973 | – | **626** | *(-35.66%)* |
| clang-31259 | 3,086 | **1,079** | *(-65.04%)* | 3,086 | – | **1,079** | *(-65.04%)* |
| gcc-59903 | 6,172 | **1,825** | *(-70.43%)* | 6,172 | – | **1,825** | *(-70.43%)* |
| gcc-60116 | 4,300 | **1,810** | *(-57.91%)* | 4,300 | – | **1,810** | *(-57.91%)* |
| gcc-61383 | 3,642 | **1,449** | *(-60.21%)* | 3,642 | – | **1,449** | *(-60.21%)* |
| gcc-61917 | 2,912 | **695** | *(-76.13%)* | 2,912 | – | **695** | *(-76.13%)* |
| gcc-64990 | 2,051 | **1,094** | *(-46.66%)* | 2,051 | – | **1,094** | *(-46.66%)* |
| gcc-65383 | 1,671 | 768 | *(-54.04%)* | 1,733 | *(+3.71%)* | **763** | *(-54.34%)* |
| gcc-66186 | 4,335 | 1,221 | *(-71.83%)* | 4,349 | *(+0.32%)* | **1,219** | *(-71.88%)* |
| gcc-66375 | 5,057 | **1,848** | *(-63.46%)* | 5,057 | – | **1,848** | *(-63.46%)* |
| gcc-70127 | 2,609 | **1,472** | *(-43.58%)* | 2,609 | – | **1,472** | *(-43.58%)* |
| gcc-71626 | 178 | **177** | *(-0.56%)* | 178 | – | **177** | *(-0.56%)* |
| jerry-3299 | 159 | **94** | *(-40.88%)* | 159 | – | **94** | *(-40.88%)* |
| jerry-3361 | 108 | **95** | *(-12.04%)* | 108 | – | **95** | *(-12.04%)* |
| jerry-3376 | 72 | **39** | *(-45.83%)* | 72 | – | **39** | *(-45.83%)* |
| jerry-3408 | 74 | **66** | *(-10.81%)* | 74 | – | **66** | *(-10.81%)* |
| jerry-3431 | 33 | **31** | *(-6.06%)* | 33 | – | **31** | *(-6.06%)* |
| jerry-3433 | **18** | **18** | – | **18** | – | **18** | – |
| jerry-3437 | 48 | **32** | *(-33.33%)* | 48 | – | **32** | *(-33.33%)* |
| jerry-3479 | 165 | **111** | *(-32.73%)* | 165 | – | **111** | *(-32.73%)* |
| jerry-3483 | **38** | **38** | – | **38** | – | **38** | – |
| jerry-3506 | **57** | **57** | – | **57** | – | **57** | – |
| jerry-3523 | 63 | **48** | *(-23.81%)* | 63 | – | **48** | *(-23.81%)* |
| jerry-3534 | **69** | 80 | *(+15.94%)* | **69** | – | 80 | *(+15.94%)* |
| jerry-3536 | 132 | **124** | *(-6.06%)* | 132 | – | **124** | *(-6.06%)* |

**b** Number of Test Executions

| Test | Coarse HDD* | Coarse Hoist*+HDD* | | Coarse HDDH* | | Coarse Hoist*+HDDH* | |
|---|---|---|---|---|---|---|---|
| clang-22382 | **5,603** | 6,445 | *(+15.03%)* | **5,603** | – | 6,444 | *(+15.01%)* |
| clang-22704 | **4,391** | 19,809 | *(+351.13%)* | **4,391** | – | 19,809 | *(+351.13%)* |
| clang-23309 | 9,706 | **7,810** | *(-19.53%)* | 9,706 | – | **7,810** | *(-19.53%)* |
| clang-23353 | 50,489 | **26,543** | *(-47.43%)* | 50,489 | – | **26,543** | *(-47.43%)* |
| clang-25900 | **5,215** | 5,383 | *(+3.22%)* | **5,215** | – | 5,383 | *(+3.22%)* |
| clang-26350 | **7,937** | 14,378 | *(+81.15%)* | **7,937** | – | 14,378 | *(+81.15%)* |
| clang-26760 | **4,916** | 8,849 | *(+80.00%)* | **4,916** | – | 8,849 | *(+80.00%)* |
| clang-27747 | **3,209** | 11,735 | *(+265.69%)* | **3,209** | – | 11,735 | *(+265.69%)* |
| clang-31259 | 6,625 | **4,923** | *(-25.69%)* | 6,625 | – | **4,923** | *(-25.69%)* |
| gcc-59903 | 8,679 | **7,238** | *(-16.60%)* | 8,679 | – | **7,238** | *(-16.60%)* |
| gcc-60116 | **8,083** | 8,262 | *(+2.21%)* | **8,083** | – | 8,262 | *(+2.21%)* |
| gcc-61383 | **7,504** | 8,322 | *(+10.90%)* | **7,504** | – | 8,322 | *(+10.90%)* |
| gcc-61917 | 6,624 | **5,051** | *(-23.75%)* | 6,624 | – | **5,051** | *(-23.75%)* |
| gcc-64990 | 6,706 | 7,161 | *(+6.78%)* | **6,547** | *(-2.37%)* | 7,218 | *(+7.63%)* |
| gcc-65383 | 5,495 | **3,974** | *(-27.68%)* | 5,717 | *(+4.04%)* | 4,051 | *(-26.28%)* |
| gcc-66186 | **6,606** | 10,503 | *(+58.99%)* | 6,760 | *(+2.33%)* | 10,324 | *(+56.28%)* |
| gcc-66375 | **8,092** | 11,008 | *(+36.04%)* | **8,092** | – | 11,008 | *(+36.04%)* |
| gcc-70127 | **8,015** | 11,792 | *(+47.12%)* | **8,015** | – | 11,792 | *(+47.12%)* |
| gcc-71626 | **1,808** | 4,208 | *(+132.74%)* | **1,808** | – | 4,208 | *(+132.74%)* |
| jerry-3299 | **83** | 135 | *(+62.65%)* | **83** | – | 135 | *(+62.65%)* |
| jerry-3361 | **60** | 175 | *(+191.67%)* | **60** | – | 175 | *(+191.67%)* |
| jerry-3376 | **62** | 396 | *(+538.71%)* | **62** | – | 396 | *(+538.71%)* |
| jerry-3408 | **62** | 170 | *(+174.19%)* | **62** | – | 170 | *(+174.19%)* |
| jerry-3431 | **24** | 148 | *(+516.67%)* | **24** | – | 148 | *(+516.67%)* |
| jerry-3433 | **15** | 55 | *(+266.67%)* | **15** | – | 55 | *(+266.67%)* |
| jerry-3437 | **29** | 34 | *(+17.24%)* | **29** | – | 34 | *(+17.24%)* |
| jerry-3479 | **130** | 507 | *(+290.00%)* | **130** | – | 507 | *(+290.00%)* |
| jerry-3483 | **28** | 54 | *(+92.86%)* | **28** | – | 54 | *(+92.86%)* |
| jerry-3506 | **59** | 197 | *(+233.90%)* | **59** | – | 197 | *(+233.90%)* |
| jerry-3523 | **42** | 371 | *(+783.33%)* | **42** | – | 371 | *(+783.33%)* |
| jerry-3534 | **89** | 122 | *(+37.08%)* | **89** | – | 122 | *(+37.08%)* |
| jerry-3536 | **47** | 121 | *(+157.45%)* | **47** | – | 121 | *(+157.45%)* |

**TABLE A5** Coarse Hoisting and HDD$^r$

**a** Output Sizes (Number of Non-whitespace Characters)

| Test | Coarse HDD$^{r*}$ | Coarse Hoist$^{r*}$+HDD$^{r*}$ | | Coarse HDDH$^{r*}$ | | Coarse Hoist$^{r*}$+HDDH$^{r*}$ | |
|---|---|---|---|---|---|---|---|
| clang-22382 | 1,007 | **695** | *(-30.98%)* | 1,007 | – | **695** | *(-30.98%)* |
| clang-22704 | 830 | **305** | *(-63.25%)* | 830 | – | **305** | *(-63.25%)* |
| clang-23309 | 6,198 | **2,237** | *(-63.91%)* | 6,198 | – | **2,237** | *(-63.91%)* |
| clang-23353 | 884 | **462** | *(-47.74%)* | 884 | – | **462** | *(-47.74%)* |
| clang-25900 | 2,574 | **1,206** | *(-53.15%)* | 2,574 | – | **1,206** | *(-53.15%)* |
| clang-26350 | 3,658 | **803** | *(-78.05%)* | 3,658 | – | **803** | *(-78.05%)* |
| clang-26760 | 1,349 | **564** | *(-58.19%)* | 1,349 | – | 719 | *(-46.70%)* |
| clang-27747 | 995 | **714** | *(-28.24%)* | 995 | – | **714** | *(-28.24%)* |
| clang-31259 | 3,108 | **1,270** | *(-59.14%)* | 3,108 | – | **1,270** | *(-59.14%)* |
| gcc-59903 | 4,076 | **2,131** | *(-47.72%)* | 4,076 | – | **2,131** | *(-47.72%)* |
| gcc-60116 | 4,829 | **2,171** | *(-55.04%)* | 4,829 | – | **2,171** | *(-55.04%)* |
| gcc-61383 | 4,045 | **977** | *(-75.85%)* | 4,045 | – | **977** | *(-75.85%)* |
| gcc-61917 | 2,921 | **700** | *(-76.04%)* | 2,921 | – | **700** | *(-76.04%)* |
| gcc-64990 | 2,019 | 1,063 | *(-47.35%)* | 2,412 | *(+19.47%)* | **1,061** | *(-47.45%)* |
| gcc-65383 | 1,663 | **598** | *(-64.04%)* | 1,665 | *(+0.12%)* | 768 | *(-53.82%)* |
| gcc-66186 | 4,304 | **1,226** | *(-71.51%)* | 4,308 | *(+0.09%)* | 1,264 | *(-70.63%)* |
| gcc-66375 | 5,075 | **1,788** | *(-64.77%)* | 5,075 | – | **1,788** | *(-64.77%)* |
| gcc-70127 | 2,661 | **1,395** | *(-47.58%)* | 2,661 | – | **1,395** | *(-47.58%)* |
| gcc-71626 | 178 | **177** | *(-0.56%)* | 178 | – | **177** | *(-0.56%)* |
| jerry-3361 | 96 | **94** | *(-2.08%)* | 96 | – | **94** | *(-2.08%)* |
| jerry-3299 | 108 | **95** | *(-12.04%)* | 108 | – | **95** | *(-12.04%)* |
| jerry-3376 | **72** | **72** | – | **72** | – | **72** | – |
| jerry-3408 | 74 | **66** | *(-10.81%)* | 74 | – | **66** | *(-10.81%)* |
| jerry-3431 | 33 | **31** | *(-6.06%)* | 33 | – | **31** | *(-6.06%)* |
| jerry-3433 | **18** | **18** | – | **18** | – | **18** | – |
| jerry-3437 | 48 | **32** | *(-33.33%)* | 48 | – | **32** | *(-33.33%)* |
| jerry-3479 | 165 | **111** | *(-32.73%)* | 165 | – | **111** | *(-32.73%)* |
| jerry-3483 | **38** | **38** | – | **38** | – | **38** | – |
| jerry-3506 | **57** | **57** | – | **57** | – | **57** | – |
| jerry-3523 | 63 | **48** | *(-23.81%)* | 63 | – | **48** | *(-23.81%)* |
| jerry-3534 | 96 | **80** | *(-16.67%)* | 96 | – | **80** | *(-16.67%)* |
| jerry-3536 | 132 | **124** | *(-6.06%)* | 132 | – | **124** | *(-6.06%)* |

**b** Number of Test Executions

| Test | Coarse HDD$^{r*}$ | Coarse Hoist$^{r*}$+HDD$^{r*}$ | | Coarse HDDH$^{r*}$ | | Coarse Hoist$^{r*}$+HDDH$^{r*}$ | |
|---|---|---|---|---|---|---|---|
| clang-22382 | 3,157 | **2,614** | *(-17.20%)* | 3,157 | – | **2,614** | *(-17.20%)* |
| clang-22704 | **2,914** | 5,878 | *(+101.72%)* | **2,914** | – | 5,878 | *(+101.72%)* |
| clang-23309 | 4,989 | **3,384** | *(-32.17%)* | 4,989 | – | **3,384** | *(-32.17%)* |
| clang-23353 | **3,587** | 4,363 | *(+21.63%)* | **3,587** | – | 4,363 | *(+21.63%)* |
| clang-25900 | 3,096 | **2,497** | *(-19.35%)* | 3,096 | – | **2,497** | *(-19.35%)* |
| clang-26350 | **4,746** | 5,478 | *(+15.42%)* | **4,746** | – | 5,478 | *(+15.42%)* |
| clang-26760 | **2,952** | 3,047 | *(+3.22%)* | **2,952** | – | 11,602 | *(+293.02%)* |
| clang-27747 | **2,293** | 3,084 | *(+34.50%)* | **2,293** | – | 3,084 | *(+34.50%)* |
| clang-31259 | 4,031 | **2,612** | *(-35.20%)* | 4,031 | – | **2,612** | *(-35.20%)* |
| gcc-59903 | 5,160 | **4,101** | *(-20.52%)* | 5,160 | – | **4,101** | *(-20.52%)* |
| gcc-60116 | 6,268 | **4,452** | *(-28.97%)* | 6,268 | – | **4,452** | *(-28.97%)* |
| gcc-61383 | 4,652 | **2,224** | *(-52.19%)* | 4,652 | – | **2,224** | *(-52.19%)* |
| gcc-61917 | 3,875 | **2,783** | *(-28.18%)* | 3,875 | – | **2,783** | *(-28.18%)* |
| gcc-64990 | 4,805 | **2,640** | *(-45.06%)* | 4,167 | *(-13.28%)* | 2,650 | *(-44.85%)* |
| gcc-65383 | 3,595 | **1,954** | *(-45.65%)* | 3,547 | *(-1.34%)* | 1,960 | *(-45.48%)* |
| gcc-66186 | 3,715 | 2,288 | *(-38.41%)* | 4,237 | *(+14.05%)* | **2,264** | *(-39.06%)* |
| gcc-66375 | 4,369 | **2,879** | *(-34.10%)* | 4,369 | – | **2,879** | *(-34.10%)* |
| gcc-70127 | 4,236 | **2,835** | *(-33.07%)* | 4,236 | – | **2,835** | *(-33.07%)* |
| gcc-71626 | **989** | 1,325 | *(+33.97%)* | **989** | – | 1,325 | *(+33.97%)* |
| jerry-3299 | **67** | 120 | *(+79.10%)* | **67** | – | 120 | *(+79.10%)* |
| jerry-3361 | **53** | 116 | *(+118.87%)* | **53** | – | 116 | *(+118.87%)* |
| jerry-3376 | **57** | 335 | *(+487.72%)* | **57** | – | 335 | *(+487.72%)* |
| jerry-3408 | **59** | 123 | *(+108.47%)* | **59** | – | 123 | *(+108.47%)* |
| jerry-3431 | **20** | 84 | *(+320.00%)* | **20** | – | 84 | *(+320.00%)* |
| jerry-3433 | **16** | 44 | *(+175.00%)* | **16** | – | 44 | *(+175.00%)* |
| jerry-3437 | **28** | 31 | *(+10.71%)* | **28** | – | 31 | *(+10.71%)* |
| jerry-3479 | **123** | 252 | *(+104.88%)* | **123** | – | 252 | *(+104.88%)* |
| jerry-3483 | **29** | 50 | *(+72.41%)* | **29** | – | 50 | *(+72.41%)* |
| jerry-3506 | **55** | 171 | *(+210.91%)* | **55** | – | 171 | *(+210.91%)* |
| jerry-3523 | **33** | 161 | *(+387.88%)* | **33** | – | 161 | *(+387.88%)* |
| jerry-3534 | **64** | 102 | *(+59.38%)* | **64** | – | 102 | *(+59.38%)* |
| jerry-3536 | **44** | 110 | *(+150.00%)* | **44** | – | 110 | *(+150.00%)* |