# Enhancements of Automated Test Case Reduction

Ph.D. Dissertation

by

**Dániel Vince**

Supervisor:
Dr. Ákos Kiss

Doctoral School of Informatics
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged
2024

*"When you compete against everyone else, no one wants to help you. But when you compete against yourself, everyone wants to help you."*

— Simon Sinek

# Foreword

Exactly 10 years ago, in September 2014, I started my university studies. In early 2016 I was in the second year of my BSc studies when I got the opportunity to join a small .NET development team at the Department of Software Engineering. At that time, it was fascinating to see how Ph.D. students understood different topics, taught us interesting subjects, gave us programming tasks, *etc.* They were on a different level, and I wanted to achieve that. Furthermore, I also saw how my boss (and my future supervisor) traveled around the world to present his research in beautiful countries where I did not have the opportunity to travel. Then I asked my boss how I could do the same and everything changed.

From the bug-fixing tasks of the .NET projects, I started to shift to more research-related tasks, then I blinked and became a Ph.D. student. Unfortunately, Covid-19 took the wind out of my sails: I could present my work in Madrid (Spain), virtually. The next conference I could stand up was in Guangzhou (China), virtually – one bad luck after the other. I had to reposition my inner motivation, *why* should I do this? Fortunately I already had enough experience at that time to know that I love being involved in innovative teams, especially when it's obvious that the impact we can make will be noticeable by the users of our products. Back to the traveling topic, Covid-19 related restrictions ended at a time and I cannot be completely dissatisfied. I was able to visit Singapore and Vienna, those were my trips so far during the five years of my PhD studies.

*Dániel Vince, 2024*

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Our modern lives are driven by software. This has become so common that we often do not even notice the programs around us; they collect and process data and then tell us to get up and walk around for a few minutes during our long office hours. In an ideal world, the software is perfect, and the business logic fully covers the customer's expectations and produces the right output for every possible combination of inputs, so it "cannot be fooled".

Unfortunately, we do not live in an ideal world. Software is still mostly written by humans, and humans can make mistakes despite their best intentions. Nowadays, the use of code snippets generated by artificial intelligence is on the rise (*e.g.*, GitHub Copilot[1] or ChatGPT[2]), however, these cannot be blindly trusted. At least not yet. We can expect that even the most carefully designed and implemented software will contain hidden bugs that will surface sooner or later. Most likely at the worst possible time. And then, we will notice – or worse, remember – the software, and its failure.

Finding a bug can be the result of targeted human activity, or it can be the result of automated testing techniques. Whatever the source, in most

[1]https://github.com/features/copilot
[2]https://chat.openai.com

cases the goal of the testing agent is to find as many bugs as possible in the shortest amount of time. Due to the nature of this objective, when a new bug is found, the sequence of events that can reproduce it is noisy, *i.e.*, it contains both relevant and irrelevant information related to reproduction. This is not necessarily a problem from their point of view; the task has been successfully completed: a new bug has been discovered in the system, and a reproduction package is available that can be used to implement the fix.

From another point of view, however, it is really important to note how much irrelevant information is included in the description of a reported bug. Once it is reported, some lucky software engineer is given the task of fixing it. First, the description has to be interpreted, and then the bug has to be reproduced. After that, since the manifestation (bad output, incorrect display, or even a complete crash) is only a symptom, the real source has to be found so that the fix can be implemented. If most of the events required for reproduction are irrelevant, the time required to find the root cause of the malfunction can increase significantly. However, manually selecting the events that are absolutely necessary for reproduction is also a time-consuming task. There seems to be no good way for engineers to do this.

Fortunately, error-causing events ("test cases" from now) can be minimized automatically, which was already recognized in the early 2000s. The discipline of automatic test case reduction researches algorithms that can reduce any kind of input to a smaller, some kind of "minimal" form, while maintaining a well-defined condition. One of the most well-known algorithms is the minimizing Delta Debugging algorithm [39] (DDMIN), which tries to produce a minimal version of its input regardless of its structure while keeping a predefined property invariant. It decomposes the input into atomic units (*e.g.*, lines or characters for textual input) and systematically tries to remove pieces from it.

With the spread of programs that expect structured input, more efficient algorithms have been proposed to minimize test cases by taking structure into account. For example, programming or markup languages have a well-defined structure, but anything that can be defined with a context-free grammar is suitable. Compilers (*e.g.*, GCC and Clang) or execution engines (*e.g.*, v8, JerryScript, and Python) can only be exhaustively tested if the input passes the syntactic parser. Grammars can be used to build a tree representation of the test case, which can be reduced more efficiently while preserving the structural boundaries. One of the most well-known grammar-based algorithms is Hierarchical Delta Debugging [25] (HDD), which traverses the tree with a breadth-first search and then applies DDMIN to its levels.

Both mentioned algorithms are discussed in this work. Several functional and non-functional properties and their improvements are detailed in the following thesis. Some of them result in smaller outputs, while others aim to reduce the memory footprint or the time required by the process.

The rest of this dissertation is organized as follows: First, Chapter 2 discusses the DDMIN and HDD algorithms, their variants, and related work. Chapter 3 provides evaluation details, the used test suites and their properties, and the open-source projects on which the discussed optimizations are based. Chapter 4 describes optimizations that do not affect the output itself, but help the algorithms be more resource-efficient. A fixed-point iteration of DDMIN is presented in Chapter 5, which aims at a more effective reduction. A greedy parallel algorithm variant of DDMIN is presented in Chapter 6, and then a new transformation-based reduction method inspired by HDD is presented in Chapter 7. Finally, the results and important conclusions are summarized.

# 2 Background and Related Work

## 2.1 Minimizing Delta Debugging

The minimizing Delta Debugging (DDMIN) algorithm [15, 39, 40] is a systematic iterative approach for reducing a test case while keeping an interesting property invariant. The input of the algorithm is 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 to see 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 case parts keep 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.

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

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

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

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

**Figure 2.1:** *The Minimizing Delta Debugging algorithm.*

The algorithm has its roots in the isolation of failure-inducing code changes, which is visible in its terminology. For the algorithm, an input is composed of elementary changes or deltas, denoted as $\delta_i$, from which the algorithm got its name. A set of changes is also called a configuration, usually denoted by $c$. The outcome of a program running on a specific configuration is determined by a testing function, and it can be *fail* (also written as ✗) if the test case induced the original failure, *pass* (also written as ✓) if the test succeeded or *unresolved* (written as **?**) if the result is indeterminate. From a practical perspective, the *unresolved* outcome is treated as a *pass*, since the configuration definitely did not reproduce the original failure.

The set of all changes, *i.e.*, the initial configuration that triggers a failure is denoted by $c_{\text{✗}}$. 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 “interestingness” of a test case is not a program failure, most authors follow the original notation for historical reasons. Figure 2.1 gives Zeller and Hildebrandt’s latest formulation of the minimizing Delta Debugging algorithm [40].

Since its publication, DDMIN has enjoyed the undivided attention of researchers, especially since security-related fuzz testing became popular. Therefore, several algorithm variants and general optimizations were proposed, which are discussed in more detail in Section 2.4. However, a few ideas should be discussed here, as the optimizations presented later in this chapter are built upon them.

Hodován *et al.* [16, 18] investigated how efficient DDMIN is and proposed several practical improvements to it. They noticed that the “reduce to subset” and “reduce to complement” phases are interpreted sequentially since the first appearance of the algorithm, however, $\exists i \in \{1, \ldots, n\}$ in Figure 2.1 does not specify how to iterate that $i$. They rewrote $ddmin_2$ to use parallel loops in a way that fully utilizes the parallelization capabilities of modern systems. Their formalization left minor details to the implementation; however, one major aspect is important in this study: if the parallel loops are started and one of them triggers the failure, then the other loops are aborted, even if they have not finished yet. This technique might cause the test results to be thrown away, but the intention was to make the reduction as fast as possible, and this does not violate the integrity of the 1-minimality. They also observed that the “reduce to subset” trials are mostly greedy attempts to bite the most from the configuration with the least amount of effort, however, this code path is not necessary for 1-minimality. Programming languages were among their interests, and they realized that keeping just the middle part of a test case results in a syntactically incorrect configuration (in most cases), thus unable to reproduce the desired behavior. Therefore, they proposed to completely omit the “reduce to subset” case.

## 2.2 Hierarchical Delta Debugging

If a test case 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) algorithm [25] is to avoid such superfluous steps by not testing format-breaking configurations. To achieve this goal, it works on hierarchical tree-structured input representations (*e.g.*, on parse trees, abstract syntax trees, or XML DOM trees) and applies the DDMIN algorithm to the levels of the tree, progressing downward from the root to the leaves.

The pseudocode formulation of HDD as defined by Misherghi and Su [25] is shown in Figure 2.2(a). 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 the removal of a node causes the

```
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)** *Hierarchical Delta Debugging.*

```
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)** *Recursive Hierarchical Delta Debugging.*

**Figure 2.2:** *The Hierarchical Delta Debugging algorithm variants.*

removal of the entire subtree rooted at that node. In a later algorithm variant, the “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 [26]. 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 no longer be interesting.

Several variants have been proposed since the original definition of HDD, two of which are worth discussing here: the recursive (HDD$^r$[1]) and the coarse Hierarchical Delta Debugging algorithms (Coarse HDD). The idea behind HDD$^r$ [24] is to pass only related parts of the tree to DDMIN to ensure that it does not create partitions that cross the boundaries of the subtree (which often leads to superfluous steps). Therefore, HDD$^r$ applies DDMIN not to all nodes at a given level of the tree, but only to the sibling nodes. 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 2.2(b). The auxiliary routines *tagChildren* and *pruneChildren* differ from their *tagNodes* and *prune* counterparts only in the set of nodes on which they work, *i.e.*, on the children of a given node instead of all nodes at a given level. HDD$^r$ has the same theoretical minimality guarantees as HDD.

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

```
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
```

**(a)** *Coarse Hierarchical Delta Debugging.*

```
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
```

**(b)** *Coarse Recursive Hierarchical Delta Debugging.*

**Figure 2.3:** *The Coarse Hierarchical Delta Debugging algorithm variants.*

Coarse HDD [19] 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 2.3(a), visits all levels of the tree like the original HDD but filters out those nodes from the 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 $\Phi$ [26]). 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, then the main HDD algorithm minimizes the preprocessed tree faster. It should be noted that there is a potential variant of HDD that has not been defined in the literature, which is the combination of coarse and recursive ideas, using the recursive iteration approach while focusing on particular nodes only. Coarse HDD$^r$ is formally defined in Figure 2.3(b).

## 2.3 Caching Solutions

Zeller [39] has already raised the issue that testing an arbitrary configuration may take time. If the input to be tested is a program in a source code form, its recompilation and re-execution could take seconds, minutes, or even hours, and this time can be considerably reduced by smart recompilation techniques. Even if no recompilation is needed (*e.g.*, providing XML files to an XML parser or JavaScript inputs to an execution engine), program execution can take a long time. Due to the greedy nature of the discussed algorithms, it might happen that the same configuration is tested multiple times among the iterations. Therefore, all of them can utilize cache memory to improve their execution time at the cost of increased memory usage. However, the concept of caching is orthogonal to the algorithms themselves, and studies have mainly focused on the latter.

To avoid running the same test twice, Zeller provided an *outcome caching* mechanism in his reference implementation[2]. The cache is implemented as a tree structure, where each node is labeled with a unit of the configuration and an outcome that corresponds to the subconfiguration formed by the units from the current node up to the root of the tree. If the following configurations are in the cache: $(\langle 1, 2\rangle, ✗)$, $(\langle 1, 2, 3\rangle, ✓)$, and $(\langle 1, 4, 5\rangle, ✗)$, then they can be represented in a tree structure as shown in Figure 2.4. If a configuration has already been tested, there is a path from the root to the node along with the labels, and the end of the path contains the result of the testing function (✗, ✓, ?). When the algorithm creates a new configuration, a cache lookup is performed first. If the lookup succeeds, the previously determined test outcome is returned without the need for an actual (and potentially long-lasting) test execution. Otherwise, the configuration is tested, and the outcome is inserted into the cache. (When inserting outcomes in the cache, inner nodes may be added to the tree that represent configurations that have not been tested yet. These nodes are not labeled with an outcome at that point of the algorithm but may get an outcome assigned later on. Figure 2.4 shows $\langle 1, 4\rangle$ and $\langle 1\rangle$ as examples of such not yet tested configurations.)

Although the above-described configuration-based cache is sufficiently efficient with DDMIN, it may not be the best approach for HDD. Hodován *et al.* formalized this problem in their study [20]: various configurations of tree nodes at a given level may produce the same serialized output, configurations on different levels may induce the same output; furthermore, configurations

[2]https://www.st.cs.uni-saarland.de/dd/DD.py

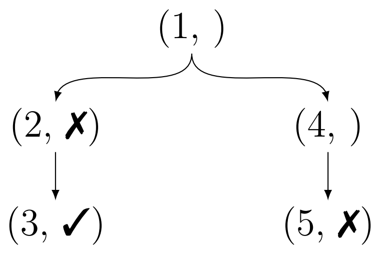


**Figure 2.4:** *The outcome caching approach of Delta Debugging.*

of different HDD* iterations may also produce the exact same output. All such configurations yield the same test outcome as well. If the outcome cache is based on tree nodes of a given level, none of these recurrences would be detected, *i.e.*, they will result in cache misses and require repeated test attempts. Motivated by these insights, they proposed to optimize HDD by using a content-based cache, *i.e.*, storing the serialized test case instead of the configuration as a key and the test outcome as the value. Therefore, if multiple configurations yield the same test case, this type of cache avoids the duplicated testing steps. The content-based cache is an optimization motivated by HDD, however, it works with DDMIN as well.

## 2.4 Related Work

One of the first works on automated test case reduction is minimizing Delta Debugging by Zeller and Hildebrandt [15, 39, 40], minimizing inputs of arbitrary format. The authors have already recognized that the same configuration may be tested at different stages of the reduction; thus, they provided a configuration-based outcome caching solution in their reference implementation. Gharachorlu and Sumner [10] observed that after a successful "reduce to complement" step, DDMIN revisits the previously investigated subsets, and these steps might be inefficient in practice. Therefore, they proposed a modified version of DDMIN, the *One Pass Delta Debugging (OPDD)*, which skips these steps, and showed that if certain circumstances are met, it can also achieve a 1-minimal result. Their goal was to achieve the linear time complexity; therefore, they identified three independent conditions that can eliminate the need for revisiting: common dependence order, unambiguity, and deferred removal.

Hodován and Kiss proposed to use DDMIN in a parallel way that can reduce the time required to perform the minimization [18]. They observed that parallel DDMIN can yield different-sized but still 1-minimal outputs as a result of independent parallel executions. Furthermore, they also observed that the "reduce to subset" step is not even necessary for 1-minimality, and reordered the "reduce to complement" steps. Kiss [23] proved that if the split factor of DDMIN is well chosen (2 was used in Zeller's work), then the reduction can be sped up significantly.

Artho [1] investigated Delta Debugging in his "Iterative Delta Debugging" study. Although the title of the study and a methodology discussed later in this chapter are similar, the two studies are not related. It used the Delta Debugging algorithm (not DDMIN) to find the failure-inducing changes in version histories. He raised the issue that DD is only applicable if the version that *passes* a test is known, which may not be the case for newly discovered defects. Therefore, he proposed the iterative DD, called IDD, that successively backports fixes to earlier defects, and one may eventually obtain a version that is capable of executing the test in question correctly.

Most of the published works target textual inputs; however, 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.* [28] minimized event sequences of distributed systems, Bársony [2] reduced OpenGL API traces, and Clapp *et al.* [9] aimed at Android GUI events with a variant of DDMIN. Furthermore, Brummayer and Biere [7] even used DDMIN to minimize SMT solver formulas.

The price of DDMIN's generality for structured inputs is a potentially lowered performance because of format-breaking, thus incorrect test cases are generated and evaluated during the reduction process. To avoid syntactically broken intermediate test cases, Miserghi and Su [25] proposed using information about the format encoded in context-free grammars, *i.e.*, to convert test cases into a tree representation and apply DDMIN to the tree levels. This approach, called Hierarchical Delta Debugging, helped remove parts of the test case that aligned with syntactic unit boundaries. As a further improvement, Miserghi [26] proposed the concept of a syntactically correct replacement for nodes that cannot be completely removed from the test case without causing syntax errors. 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.* [17] suggested using extended context-free grammars for building the tree, creating more balanced representations, which could lead to smaller results and improved performance. They have also described various tree transformations with the same goal [19, 20].

Tree-based test case reduction does not necessarily mean subtree removal. Bruno [8] suggested using hoisting as an alternative transformation in his framework called SIMP, 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 [27] extended SIMP with Delta Debugging. The main algorithm was hoisting, while DDMIN was applied only to repeated structures, such as lists (column names) and data (string literals). Sun *et al.* [30] combined the above approaches into their Perses framework. In their work, they utilized quantifiers and normalized the parse tree producing grammars by rewriting recursive rules to use quantified expressions, and the transformed grammar form was referred to as Perses Normal Form (PNF). During the reduction, they applied a worklist algorithm, in which nonterminals 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 nonquantified, regular ones. Based on the ideas introduced in Perses, Gharachorlu and Sumner [11] extended it in a new framework, named Pardis, with an improved queue prioritization algorithm. Pardis only considered completely removable nodes and assigned weights based on a node's own token weight instead of its parents. They completely eliminated the hoisting step, since they found it too expensive from a performance perspective. Herfert *et al.* [14] also combined subtree removal and hoisting in their Generalized Tree Reduction (GTR) algorithm. The applicability of a particular tree transformation was learned from an existing test corpus without analyzing a grammar.

Binkley *et al.* [3, 4, 5, 6, 12, 38] recognized an interesting analogy between test case reduction and program slicing. They have realized that the concepts of slicing can be reformulated as concepts of test case reduction. 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. Stepanov *et al.* [29] suggested a combined approach using program slicing, HDD, and Kotlin-specific transformation in their "ReduKtor" prototype tool. To avoid rechecking, they hashed the AST configurations and stored them together with the outcome. Using AST configurations for cache lookups, either in full or in hashed form, can face the same problem as traditional configuration-based caching, *i.e.*, different AST configurations can produce the same serialized test case.

# 3
# Evaluation Details

The following chapters present different algorithm optimizations, and we wanted to evaluate them on real-life scenarios uniformly (*i.e.*, with JavaScript execution engines and C/C++ compilers that are continuously tested). In order not to be repetitive, the common parts of the evaluation have been organized into this separate chapter. The following information describes the used test suites, their structure and important properties. Then, the open-source projects are discussed in which we implemented our optimizations.

As inputs, test cases from different sources have been collected, some of which have already been used in the literature for benchmarking reduction. The first test suite is the Perses Test Suite[1] (PTS), which contains fuzzer-generated C sources that cause various internal compiler errors in the Clang and GCC compilers.

The second, newly created test set is the JerryScript Reduction Test Suite[2] (JRTS), which also contains fuzzer-generated (with the Fuzzinator[3] tool[21]) 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

[1]`https://github.com/uw-pluverse/perses`

[2]`https://github.com/vincedani/jrts`

[3]`https://github.com/renatahodovan/fuzzinator`

to keep during reduction is the failure that they induce. A typical test case contains two different elements:

- *input_file.*{*c, js*}: the input file to reduce, contains fuzzer generated constructions in the appropriate programming language,
- *oracle*: usually a bash script that takes an input_file.{c, js}, and decides whether it keeps the interesting property of the initial input (return or exit with value 0) or not.

The properties of the test cases are shown in Tables 3.1 and 3.2. *Size (bytes)* is the absolute size of the test case expressed in bytes, column *Rows* shows the number of lines, while column *Chars* expresses the number of non-whitespace characters in it. To be able to feed test cases to HDD, they have to be processed to a tree structure. *Tree Height* represents the height of the parse tree built from the input, *Rules* shows the number of nonterminals, and *Tokens* shows the number of terminals in it. The parse tree representation of each test case was built using the grammar available for the input format from the official ANTLR v4 grammars repository[4], and Picireny has applied the squeeze of the linear tree components [20] and the flattening of recursive structures [19] to the trees.

As the formats of the test cases are similarly structured and come from the same domain of programming languages, the results of the following optimizations may not generalize to all types of test cases. However, we believe that the results of these test suites are indicative since they contain real-world test cases and have been used in reduction-related studies [11, 23, 30]. By comparing their sizes, JRTS contains fewer test cases, which are also smaller in size than PTS, threatening the unbiased presentation of the results. The relative effects of the proposed optimizations are examined to avoid the misinterpretation of the results, furthermore, where the results were different, we discussed them in detail.

To evaluate the effects of the discussed optimizations, a prototype is implemented for each proposal based on the open-source Picire[5] and Picireny[6] projects. Picire is a Python implementation of DDMIN, which supports parallelization and several configuration options since it has already been used for a few studies [18, 20, 23]. Picireny is a hierarchical test case reduction framework

[4]`https://github.com/antlr/grammars-v4`

[5]`https://github.com/renatahodovan/picire`

[6]`https://github.com/renatahodovan/picireny`

**Table 3.1:** *Properties of the Perses Test Suite*

| Test | Size (bytes) | Chars | Rows | Tree Height | Rules | Tokens |
|---|---:|---:|---:|---:|---:|---:|
| clang-22382 | 80,210 | 65,786 | 2,993 | 242 | 29,344 | 6,573 |
| clang-22704 | 723,495 | 597,827 | 20,617 | 272 | 255,972 | 61,255 |
| clang-23309 | 147,879 | 118,178 | 2,815 | 288 | 52,183 | 11,570 |
| clang-23353 | 134,381 | 94,734 | 4,011 | 185 | 44,100 | 9,989 |
| clang-25900 | 328,729 | 245,065 | 5,546 | 292 | 106,751 | 23,406 |
| clang-26350 | 467,008 | 378,160 | 6,759 | 304 | 168,324 | 25,790 |
| clang-26760 | 793,470 | 588,548 | 10,104 | 340 | 288,964 | 60,762 |
| clang-27747 | 541,699 | 409,083 | 8,141 | 265 | 238,604 | 46,295 |
| clang-31259 | 179,380 | 137,161 | 2,736 | 331 | 66,291 | 14,590 |
| gcc-59903 | 217,161 | 166,754 | 3,225 | 298 | 76,531 | 17,322 |
| gcc-60116 | 326,769 | 218,223 | 13,566 | 279 | 100,651 | 21,479 |
| gcc-61383 | 142,054 | 110,643 | 4,824 | 303 | 46,786 | 9,070 |
| gcc-61917 | 343,503 | 254,742 | 13,827 | 254 | 115,834 | 24,508 |
| gcc-64990 | 554,312 | 439,587 | 6,593 | 342 | 200,107 | 45,000 |
| gcc-65383 | 158,731 | 125,221 | 2,687 | 254 | 58,846 | 13,237 |
| gcc-66186 | 177,924 | 139,087 | 2,713 | 258 | 65,228 | 14,434 |
| gcc-66375 | 248,824 | 191,827 | 3,674 | 282 | 86,512 | 19,216 |
| gcc-70127 | 540,224 | 400,556 | 7,780 | 293 | 210,039 | 44,942 |
| gcc-71626 | 18,975 | 14,465 | 724 | 20 | 8,044 | 2,047 |

on top of Picire, also written in Python, that supports ANTLR v4[7] grammars and already contains an implementation of the HDD algorithm. It has also been used in several studies [17, 19, 20, 24].

In order to ensure that the implementation of the experiments is correct and accurate, we conducted a code review. On selected C and JavaScript examples, the behavior of the implementation was traced to validate that it works as intended. Furthermore, the implementation is based on open-source and well-maintained repositories such as the Picire and Picireny frameworks that have been used in several studies, and ANTLR v4.

[7]`https://github.com/antlr/antlr4`

**Table 3.2:** *Properties of the JerryScript Reduction Test Suite*

| **Test** | **Size (bytes)** | **Chars** | **Rows** | **Tree Height** | **Rules** | **Tokens** |
|---|---:|---:|---:|---:|---:|---:|
| jerry-3299 | 1,767 | 1,208 | 54 | 33 | 608 | 140 |
| jerry-3361 | 1,953 | 1,520 | 43 | 28 | 562 | 163 |
| jerry-3376 | 6,626 | 4,647 | 178 | 36 | 2,194 | 473 |
| jerry-3408 | 2,681 | 2,100 | 44 | 28 | 778 | 228 |
| jerry-3431 | 1,065 | 648 | 36 | 30 | 527 | 130 |
| jerry-3433 | 961 | 652 | 36 | 24 | 378 | 82 |
| jerry-3437 | 6,597 | 4,623 | 178 | 36 | 2,188 | 471 |
| jerry-3479 | 5,201 | 3,998 | 95 | 25 | 1,326 | 347 |
| jerry-3483 | 492 | 326 | 18 | 19 | 193 | 48 |
| jerry-3506 | 3,760 | 2,735 | 100 | 28 | 1,278 | 343 |
| jerry-3523 | 3,928 | 2,802 | 118 | 28 | 1,416 | 345 |
| jerry-3534 | 1,927 | 1,409 | 53 | 28 | 641 | 176 |
| jerry-3536 | 829 | 592 | 27 | 23 | 310 | 71 |

*"It's better to go slowly in the right direction, than to go speeding off in the wrong direction."*

— Simon Sinek

# 4 Cache Optimizations

Although the content-based cache (see Section 2.3) improved the efficiency of reduction, there is still room for improvement. General-purpose caching techniques try to maximize the utilization of available (usually fixed-size) memory by keeping the most popular entries in the cache [22]. The most widespread algorithms for cache replacements are Least Frequently Used (LFU), Most Frequently Used (MFU), and Least Recently Used (LRU) [13], but these classic techniques do not make use of knowledge of the underlying algorithms, and evict elements from cache that might be needed later.

Consider the following example: Given an input to reduce that contains numbers from 1 to 5, one character each, and the interesting property to keep is to contain the numbers 2 and 4, then we would like to reduce the text of "12345" to the form of "24". Table 4.1 shows the character-based reduction process of DDMIN step by step. For the sake of simplicity, the "reduce to subset" steps are skipped, only the "reduce to complement" steps are presented.

As discussed in Section 2.1, the basic concept of DDMIN is that if it finds a *failing* configuration that results in a serialized test case of size $n$, then it starts a new iteration with that to reduce it further. Hereinafter, configurations that result in test cases larger than $n$ would not be tested, since the new iteration splits that configuration into smaller fragments, *e.g.*, in the 4th step in Table 4.1,

**Table 4.1:** *Execution of minimizing Delta Debugging on a Motivational Example*

| Step | Content | Action | Outcome |
|---|---|---|---|
| 1 | "12" | *test* | ✓ |
| 2 | "345" | *test* | ✓ |
| 3 | "123" | *test* | ✓ |
| 4 | "1245" | *test* | ✗ |
| 5 | "145" | *test* | ✓ |
| 6 | "245" | *test* | ✗ |
| 7 | "2" | *test* | ✓ |
| 8 | "45" | *test* | ✓ |
| 9 | "24" | *test* | ✗ |
| 10 | "2" | *cache* | ✓ |
| 11 | "4" | *test* | ✓ |

the size of the serialized test case is 4 ("1245") and that would be split further into "145" and "245" in later steps. This observation can be written as follows, using the notation introduced in Section 2:

$$\begin{gathered} c_x, c_y \subseteq c_{✗} \\ \|.\| : \text{size of the serialized configuration} \\ \exists c_x : \mathit{test}(c_x) \rightarrow ✗ \text{ found} \\ \forall c_y : \|c_y\| > \|c_x\| : \text{ out of search space} \end{gathered} \tag{4.1}$$

It is known that after a failing configuration is found, its subsets would be reduced further via DDMIN, thus theoretically there is no chance of getting a *failing* outcome back from the cache. Suppose that we have a configuration of size $n$, and before testing it, a cache lookup is performed. The cache may contain smaller entries, *e.g.*, in the 6th step in Table 4.1, where $n = 3$ and the cache already contains ("12",✓), however, if a smaller entry ($m < n$) would be in the cache with a failing outcome, then the current state could not have occurred, since DDMIN would have split that $m$ sized entry into even smaller chunks. Therefore, if a cache hit occurs, we can be sure that it was a result of a *passing* test. Thus, the *first proposal* of this chapter, as shown in (4.2), is to add only *passing* tests to the cache which may reduce the memory footprint of the minimization algorithm. Furthermore, cache lookups might be quicker since the queries are performed in a smaller search space. The function *insert to cache* inserts an element into the cache, while *serialize* performs the serialization of test cases as discussed in [20].

$$\begin{gathered} c_x \subseteq c_{\text{✗}} \\ \text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✓} \text{ found} \\ \mathit{insert\ to\ cache}\,(\ \mathit{serialize}(c_x)\ ) \end{gathered} \tag{4.2}$$

Another benefit of (4.1) is that if a failing test case is found, we can be sure that no cache entry corresponding to test cases larger than the currently found one would be queried during the remaining reduction process. Therefore, when a new failing test case is found, the entries that store the result of test cases that are larger than the currently investigated one can be evicted from the cache, as shown in (4.3). This eviction process will be referred to as the *second proposal* of this chapter. The function *delete from cache* implements the removal of an element from the cache.

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ \text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✗} \text{ found} \\ \forall c_y \colon \mathit{serialize}(c_y) \in \text{cache} \land \|c_y\| > \|c_x\| \colon \\ \mathit{delete\ from\ cache}\,(\ \mathit{serialize}(c_y)\ ) \end{gathered} \tag{4.3}$$

For small inputs, this proposal might be runtime overhead only; however, the benefits might overcome the costs for “large enough” inputs. A cache lookup is assumed to be faster than the actual test execution, but it also takes time. If the search space of the cache is maintained properly, then the time spent with lookups will be less than the time spent with eviction.

If the above-described proposals are applied, the cache will contain *passing* tests only and will be cleared after each successful reduction step. However, the lengths of the stored entries vary, *i.e.*, they are larger at the beginning of the reduction (proportional to the size of the initial failing test case) and become smaller as the process progresses towards the 1-minimum. The *third proposal* of this chapter is the following: the cache should not store the serialized content of the configurations, but their transformed form as shown in (4.4).

$$
\begin{gathered}
c_x \subseteq c_{\text{✗}}, M \colon M \in \mathbb{N} \\
\mathit{transform}(c_x) \colon 2^{\mathbb{N}} \mapsto 2^{M} \text{ bijection} \\
\text{when } \exists c_x \colon \mathit{test}(c_x) \rightarrow \text{✓} \text{ found} \\
\mathit{insert\ to\ cache}\,(\mathit{transform}(\ \mathit{serialize}(c_x)\ ))
\end{gathered}
\tag{4.4}
$$

The proposal is functional only if the transformation is bijective, *i.e.*, each test case has its own transformed form, each transformed element corresponds to exactly one test case, and unpaired elements are forbidden. From a practical perspective, the bijection is not possible, since an infinite set would have to be mapped to a finite one. Therefore, a large enough $M$ and a suitable *transform* function must be chosen to minimize the possibility of collisions of cache keys, *e.g.*, an *SHA-3-256* cryptographic hash function[1]. In contrast, if the chosen $M$ is too large, the desired positive effect on memory usage is lost.

Although the possibility of mapping two arbitrarily different test cases to the same element is negligibly small (*e.g.*, the collision resistance of an SHA-3 algorithm is $2^{n/2}$, with *SHA-3-256* it is $2^{128}$), it needs to be dealt with.

$$
\begin{gathered}
c_x, c_y \subseteq c_{\text{✗}} \\
x : \mathit{serialize}(c_x),\ y : \mathit{serialize}(c_y) \\
\exists c_x, \exists c_y \colon x \neq y \implies \mathit{transform}(x) = \mathit{transform}(y)
\end{gathered}
\tag{4.5}
$$

Suppose that $c_x$ from (4.5) has already been tested ($\mathit{test}(c_x) \rightarrow$ ✓) and inserted into the cache. Now the algorithm tries another configuration $c_y$ ($c_x \neq c_y$), performs a cache lookup, and finds that it has already been tested (since $\mathit{transform}(x) = \mathit{transform}(y)$). This state can lead to two different outcomes:

$\mathit{test}(c_x) =$ ✓ $\wedge\ \mathit{test}(c_y) =$ ✓: none of them reproduced the interesting property and the integrity of the algorithm is not compromised,

$\mathit{test}(c_x) =$ ✓ $\wedge\ \mathit{test}(c_y) =$ ✗: $c_y$ would reproduce the initial ✗, but due to the cache hit, it would never be tested.

This may lead to a suboptimal outcome, but the invariants of the algorithm are still not violated. In theory, this may lower the effectiveness of the reduction. The following Section discusses whether collisions happened in practice.

[1]https://csrc.nist.gov/projects/hash-functions/sha-3-project

## 4.1 Evaluation

The experimental setup described in Chapter 3 was used to evaluate the effects of the proposed optimizations. Beyond those settings, there are some details that are specific to this chapter: The *SHA-3-256* algorithm from the Python *hashlib* module was used for the *transform* function in (4.4), and the *pympler*[2] Python module was used to measure the cache size during reduction.

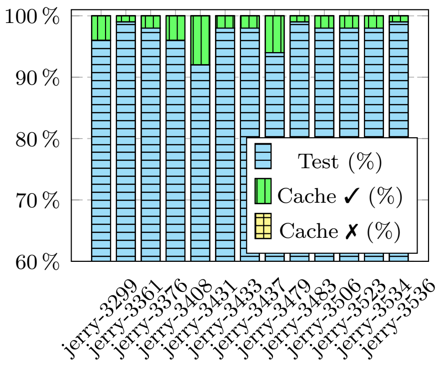
The C sources of PTS are an order of magnitude bigger than the JavaScript files of JRTS both in terms of character count and in their internal representation. This also had a negative effect on the execution time of DDMIN on tests from PTS, therefore tests from JRTS were only used for benchmarking DDMIN. For HDD, both test suites were used. To determine the effects of the proposals on the execution time, the experiments were repeated multiple times and their execution times were averaged. (The output of the reduction and the behavior of the cache were stable across the repeated experiments. The only thing that varied slightly was the execution time.)

The workstation used to conduct the experiments was 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 5.11.0, and running the experiments only.

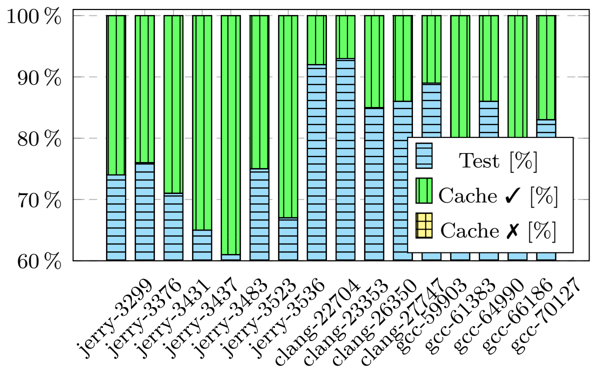
### 4.1.1 Efficiency of the Content Cache

In order to make sure the optimization ideas presented in this chapter are relevant, the behavior of the content cache should be examined first. Test cases from JRTS are reduced with DDMIN to see how many times the cache was queried and how cache hits (✓ or ✗) relate to one another. Figure 4.1(a) shows the results: the horizontally striped (blue) bars representing the test executions show that in the majority of cases (97% on average) the algorithm had to test the configuration to determine its outcome. The remaining cases are successful cache lookups: vertically striped (green) bars show cache hits with *pass* outcomes, while (yellow) bars with grid patterns stand – or, would stand – for *fail* outcomes returned from the cache. Note that, supporting (4.1), no cache lookup returned a ✗ outcome. Similar observations can be made about HDD (see Figure 4.1(b)), however, the cache is utilized better compared to DDMIN: 79% of the configurations were tested and 21% of them had outcomes in the cache, on average.

[2]https://pypi.org/project/Pympler/



**(a)** *DDMIN*

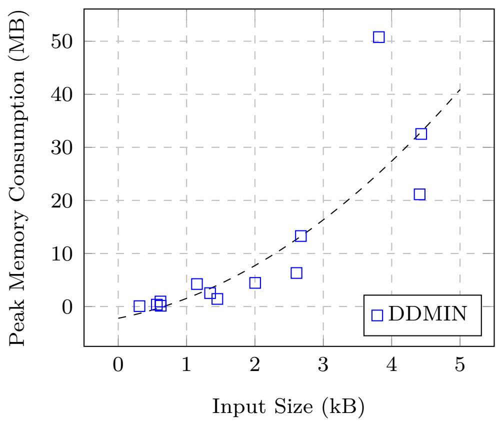


**(b)** *HDD*

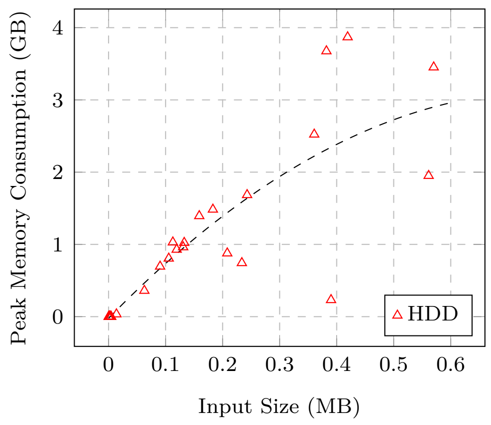
**Figure 4.1:** *Cache hits and test executions with reduction algorithms.*

We had one unexpected finding with HDD though: 9 cache lookups (0.01% of all cases) returned a ✗ result. After manual analysis of the steps of the algorithm on the test cases, it was found that this may happen when the minimal replacement of a tree node is identical to its serialized form, *i.e.*, when it does not matter if such a node is pruned or not, the same character sequence would be serialized from it. Therefore, contrary to (4.1), there is a chance of retrieving a *fail* outcome from the cache with the HDD algorithm if minimal replacements are used, even if that chance is really small. If Proposal 1 was applied when using HDD, these configurations had to be tested again, thus the required testing steps would increase by 0.01% (on the test suites used in the experiments).

The memory consumption of the cache highly depends on the size of the input both with DDMIN and HDD, as shown in Figure 4.2(a) and 4.2b. The horizontal axes show the input size (in kB and MB, respectively) and the vertical axes show the peak memory consumption (in MB and GB). Figure 4.2(a) shows how DDMIN reduced the inputs taken from JRTS. The cache could consume a relatively large amount of memory (up to 53 MB) even for small inputs (up to 4.6 kB). Figure 4.2(b) shows the same information for HDD, where tests from both JRTS and PTS are reduced. The rule of thumb is that bigger inputs cause higher memory consumption, which generally holds for both DDMIN and HDD (with the exception of some outliers). The peak memory consumption could easily reach 4 GB with HDD.



(a) *DDMIN*



(b) *HDD*

**Figure 4.2:** *Memory consumption of the content cache.*

According to these results, even though only a small percentage of the lookups resulted in a cache hit (3% with DDMIN and 21% with HDD), the cache consumed a high amount of memory. Thus, DDMIN-based reduction techniques could benefit from more efficient cache utilization.

### 4.1.2 The Effects of Optimizations

This subsection presents experimental results on how different optimizations affect the reduction process. For DDMIN, the effects of the optimizations are presented incrementally. *I.e.,* the effects of the 1st proposal are presented against the baseline, then the effects of the combined 1st and 2nd proposals are compared to the results of the 1st, and finally, it is shown how all three proposals compare against the combination of the 1st and 2nd. Technically, as described in Section 4, the proposals are considered as steps. For HDD, the results are presented for all three proposals combined.

#### Proposal 1

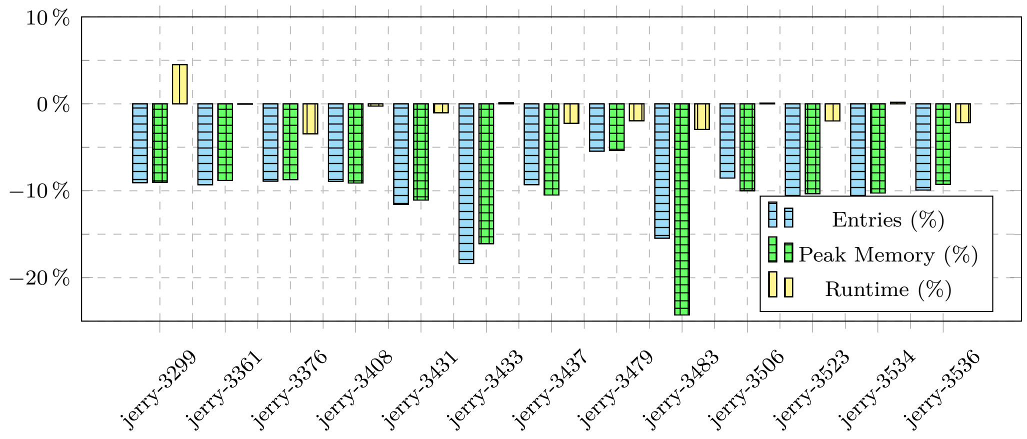
The first proposal is to avoid adding configurations to the cache that have *failing* outcomes. Figure 4.3(a) shows relative differences in the number of cache entries (horizontally striped blue bars), peak memory consumption (vertically striped green bars), and runtime (yellow bars with grid pattern), where the effects of the proposal are compared to the baseline cache implementation that stores all outcomes in the cache. The number of entries in the cache has been decreased by 10.51% on average, and by 18.37% in the best case (meaning that DDMIN finds *failing* configurations in about every tenth attempt). Likewise, on average, 11% less memory was needed to accomplish the reduction, with a 24.29% improvement in the best case. When investigating the execution time, the change is not consistent: in the best case, 3.45% of the execution time is saved (jerry-3376), but there can also be an increase, with a maximum of 4.51% (jerry-3299). The reduction is not changed beyond these characteristics, *i.e.*, the output is exactly the same as before applying the proposal.

## Proposal 2

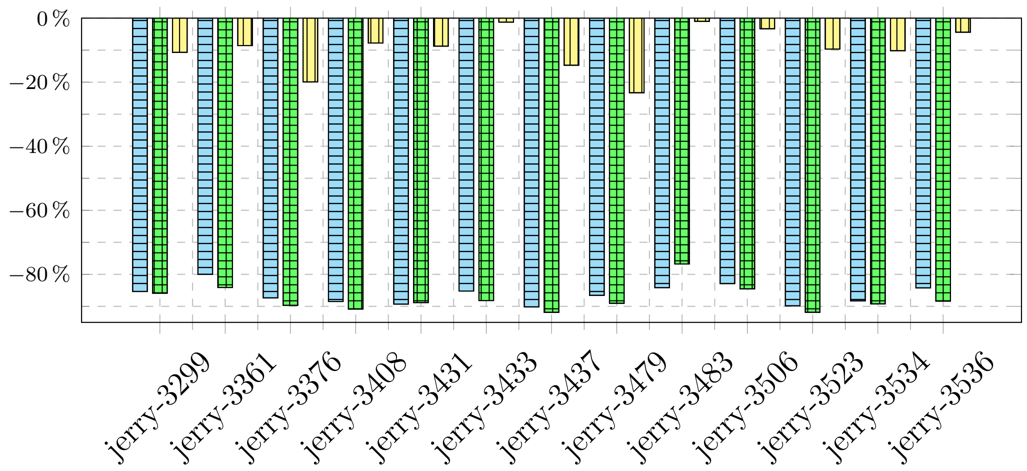
The second proposal is to do regular housekeeping and clear entries from the temporary storage that are bigger than the actually found *failing* test case. The first proposal is about not doing something, however, actively managing the cache during reduction might take additional time in exchange for reduced memory consumption. Figure 4.3(b) shows the surprising effects of this proposal from a runtime point of view: it consistently speeded up the reduction by 9.5% on average (and by 23.3% in the best case). The number of maximum cache entries got reduced by 86.29% on average (by 90.16% in the best case) and also on average, 87.63% less memory was required to finish the task. The outcome of the reduction remained the same as before applying the proposal. (Proposal 1 was considered as the reference in this comparison.)

## Proposal 3

The third proposal is about storing a transformed version of the serialized test case in the cache. As mentioned before, the *SHA-3-256* hashing algorithm was investigated to transform the test cases. However, using the hash is not compatible with Proposal 2, since the size information of the stored entries is lost. Table 4.1 presented an example algorithm execution, and the values from the “Content” and “Outcome” columns are stored in the cache as key-value pairs. The outcome is redundant information after Proposal 1 since only the *passing* test cases are stored for further usage. Thus, the size information can be stored as a value in place of the outcome, therefore, the key is the SHA-3-



**(a)** *Effects of Proposal 1 on DDMIN compared to the baseline [20]*



**(b)** *Effects of Proposal 2 on DDMIN compared to Proposal 1*

**Figure 4.3:** *Effects of the first two proposals on cache memory consumption.*

256 transformed content of the test case and the value is its size (before the transformation).

The first examined question was raised at the end of Section 4, *i.e.*, whether collisions happened during reduction: during the experiments with the used test suites and algorithm implementations, no hash collision occurred at all.

Since only the form of the stored entities changed, Figure 4.4 contains memory consumption and runtime changes only. The effect of this proposal on runtime is similar to Proposal 1, *i.e.*, relatively small changes could be observed in both directions (0.39% increase on average and +1.46% maximum). The memory consumption after applying this proposal has dropped by 65% on average and by 91.24% in the best case.

The backing data for the experiments are shown in Table 4.2. The "Baseline" column shows the peak memory consumption required for reducing the input (in kilobytes). Then, the "Proposal 1", "2", and "3" columns show the same results after applying each proposal incrementally, and the "Difference" column shows the relative difference between the baseline and the last proposal (that includes all discussed optimizations). It can be seen that after applying the proposals, the reduction process can work with a fraction of the initial memory footprint (reduced from 53.2 MB to 483.5 kB in the most extreme case). As a side effect, the execution time is improved as well. Although only Proposal 2 resulted in a

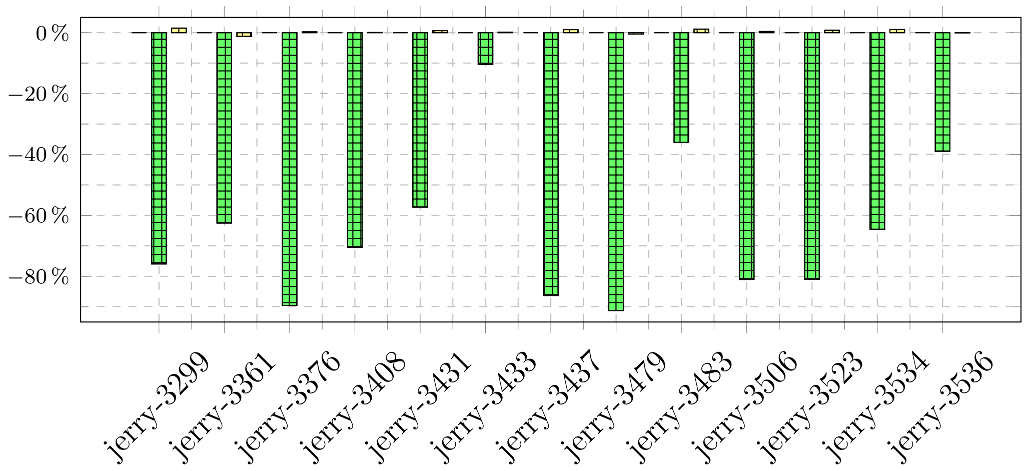


**Figure 4.4:** *Effects of Proposal 3 on DDMIN compared to Proposals 1 and 2 combined.*

**Table 4.2:** *Peak Memory Footprint of Content Cache with DDMIN*

| **Test** | **Baseline (kB)** | **Proposal 1 (kB)** | **Proposal 2 (kB)** | **Proposal 3 (kB)** | **Difference (%)** |
|---|---:|---:|---:|---:|---:|
| jerry-3299 | 4,338.4 | 3,946.7 | 556.9 | 134.1 | *-96.91%* |
| jerry-3361 | 1,461.5 | 1,332.5 | 211.7 | 79.4 | *-94.57%* |
| jerry-3376 | 33,301.8 | 30,398.5 | 3,138.4 | 328.7 | *-99.01%* |
| jerry-3408 | 4,558.0 | 4,142.5 | 378.9 | 112.0 | *-97.54%* |
| jerry-3431 | 970.6 | 863.1 | 96.5 | 41.2 | *-95.75%* |
| jerry-3433 | 145.8 | 122.3 | 14.5 | 12.9 | *-91.12%* |
| jerry-3437 | 21,652.0 | 19,381.8 | 1,578.3 | 215.9 | *-99.00%* |
| jerry-3479 | 51,993.7 | 49,201.1 | 5,392.9 | 472.1 | *-99.09%* |
| jerry-3483 | 75.3 | 57.0 | 13.2 | 8.5 | *-88.74%* |
| jerry-3506 | 6,470.1 | 5,823.0 | 900.6 | 170.7 | *-97.36%* |
| jerry-3523 | 13,604.9 | 12,198.1 | 993.7 | 188.9 | *-98.61%* |
| jerry-3534 | 2,589.7 | 2,323.9 | 250.3 | 88.8 | *-96.57%* |
| jerry-3536 | 361.3 | 327.8 | 38.1 | 23.3 | *-93.55%* |

consistent change, the average improvement after the optimizations is 9.89%.

## Hierarchical Delta Debugging

HDD uses DDMIN as a utility to minimize the nodes of its parse tree (see Figure 2.2(a)), therefore, the combined impacts of the optimizations are discussed (as the utility is replaced by the *improved DDMIN*).

When investigating experimental results from JRTS, the cache had to store 47.47% fewer entries after enabling all of the optimizations, and this required 63.19% less memory on average. Consistent trends are not present in the reduction time, but on average 0.57% more time was needed for the reduction. Measurements with PTS showed bigger improvements: on average, 86.34% fewer cache entries were stored, which resulted in a 99.93% smaller memory footprint and a 7.89% shorter runtime. As shown in Tables 3.1 and 3.2, the characteristics of the used test suites are quite different, and it can be observed that the baseline solution by Hodován *et al.* [20] does not scale well. The bigger the input, the more resources are needed to perform the reduction. Averaging the relative changes from both test suites, optimizations enabled reducing test cases with an 85% smaller memory footprint in a 4.46% shorter time. The backing data can be found in Table 4.3; the “Proposals” column shows results after applying all three proposals.

**Table 4.3:** *Peak Memory Footprint of Content Cache with HDD*

| **Test** | **Baseline (kB)** | **Proposals (kB)** | **Difference (%)** |
|---|---:|---:|---:|
| clang-22382 | 377,257.05 | 250.66 | *-99.93%* |
| clang-22704 | 3,619,547.48 | 219.04 | *-99.99%* |
| clang-23309 | 1,079,463.82 | 898.76 | *-99.92%* |
| clang-23353 | 728,406.59 | 315.23 | *-99.96%* |
| clang-25900 | 779,289.02 | 477.48 | *-99.94%* |
| clang-26350 | 2,644,713.97 | 486.09 | *-99.98%* |
| clang-26760 | 2,044,780.89 | 202.71 | *-99.99%* |
| clang-27747 | 242,636.81 | 167.38 | *-99.93%* |
| clang-31259 | 1,009,121.59 | 591.93 | *-99.94%* |
| gcc-59903 | 1,461,098.60 | 289.84 | *-99.98%* |
| gcc-60116 | 920,202.62 | 943.73 | *-99.90%* |
| gcc-61383 | 844,987.29 | 428.65 | *-99.95%* |
| gcc-61917 | 1,766,635.98 | 315.82 | *-99.98%* |
| gcc-64990 | 3,900,100.63 | 680.55 | *-99.98%* |
| gcc-65383 | 893,910.29 | 590.15 | *-99.93%* |
| gcc-66186 | 1,030,536.05 | 633.49 | *-99.94%* |
| gcc-66375 | 1,555,498.32 | 792.48 | *-99.95%* |
| gcc-70127 | 3,853,617.25 | 374.30 | *-99.99%* |
| gcc-71626 | 35,775.61 | 180.52 | *-99.50%* |
| jerry-3299 | 61.21 | 15.70 | *-74.36%* |
| jerry-3361 | 36.32 | 10.96 | *-69.82%* |
| jerry-3376 | 61.56 | 10.64 | *-82.72%* |
| jerry-3408 | 34.43 | 16.73 | *-51.40%* |
| jerry-3431 | 8.82 | 5.58 | *-36.76%* |
| jerry-3433 | 4.20 | 1.54 | *-63.38%* |
| jerry-3437 | 30.16 | 4.29 | *-85.78%* |
| jerry-3479 | 161.83 | 13.23 | *-91.83%* |
| jerry-3483 | 10.45 | 7.83 | *-25.06%* |
| jerry-3506 | 33.38 | 8.55 | *-74.37%* |
| jerry-3523 | 28.30 | 12.68 | *-55.20%* |
| jerry-3534 | 39.91 | 19.66 | *-50.75%* |
| jerry-3536 | 45.30 | 18.10 | *-60.04%* |

## 4.2 Conclusions

The caching solutions of DDMIN and HDD are investigated in this chapter. The "content-based" caching technique [20] was chosen as our baseline. Based on the experimental data and observations above, we can conclude the contributions of this chapter:

1. The cache utilization and scaling are suboptimal: DDMIN determined the outcome of its configurations via cache memory only in 3% of the cases, while HDD utilized the cache better, the actual testing of 21% of the configurations could be avoided. It did not scale well for either algorithm: DDMIN consumed up to 53 MB of memory for reducing a 4 kB sized input, while HDD required 4 GB of RAM to reduce a 0.44 MB sized test in the worst case.

2. Three optimizations were proposed to reduce the memory footprint of caches used in test case minimization:

   (a) add only *passing* (✓) tests to the cache,

   (b) when a new *failing* (✗) test case is found, evict cache entries of bigger test cases, and

   (c) instead of storing the serialized test contents, store their *hashed* value (fixed-width keys instead of variable-width).

3. With the optimizations combined, DDMIN requires 96% and HDD requires 85% less memory compared to the baseline implementation. Supporting the scalability issue, the size of the input had an effect on the results: on JRTS (smaller tests), the average improvement was 63.19%, while on PTS (larger inputs), it was 99.93%. Furthermore, as a side effect, the reduction becomes faster by 9.9% with DDMIN. In our experiments, the result of the reductions did not change after the optimizations.

*"The goal is not to be perfect by the end. The goal is to be better today."*

— Simon Sinek

# 5 Iterating the Minimizing Delta Debugging Algorithm

The program in Figure 5.1 is a variant of a classic example of program slicing [31]. It computes both the sum and the product of the first ten natural numbers in a single loop. Using slicing terminology, we can say that we want to compute the (so-called static backward) slice of this program with respect to the criterion (19, *prod*), thus creating a sub-program that does not contain statements that do not contribute to the value of *prod* at line 19.

This can be computed either by analyzing the control and data dependencies of the program – which is the classic slicing way – or by following the approach of observation-based slicing [3] that performs a systematic removal of code parts based on trial and error, much like what DDMIN does on its input. Actually, even DDMIN can be applied to such tasks. The two things that have to be remembered are that in such reduction scenarios, the inputs or test cases are also programs, and the interesting properties to keep are not program failures (but it is still an ✗ that represents that the property is kept). So, we reformulate the classic slicing example as a test case minimization task, where the program in Figure 5.1 is the input (the lines being the units), and the testing function

```
int add(int a, int b)
{
    return a + b;
}
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int sum = 0;
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        sum = add(sum, i);
        prod = mul(prod, i);
    }
    printf("sum: %d\n", sum);
    printf("prod: %d\n", prod);
}
```

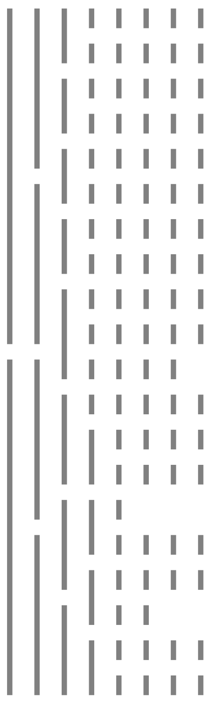


**Figure 5.1:** *Example C program that computes the sum and product of the first ten natural numbers, and the execution of DDMIN on it while keeping 10! on the output.*

is given as

$$test(c) = \begin{cases} \checkmark & \text{if } c \text{ is syntactically incorrect} \\ \checkmark & \text{else if execution of } c \text{ does not terminate} \\ \checkmark & \text{else if execution of } c \text{ does not print } \textit{prod: 3628800} \\ \text{✗} & \text{otherwise.} \end{cases}$$

The gray bars on the right of the program code show the progress of DDMIN, from left to right. Every set of vertically aligned bars corresponds to a configuration of the algorithm and shows how that configuration is split into subsets. This example shows that DDMIN could “slice away” the lines of the *main* function that did not contribute to the computation of *prod*. However, the algorithm could not remove the *add* function, because when the configuration

```
int add(int a, int b)
{
    return a + b;
}
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        prod = mul(prod, i);
    }
    printf("prod: %d\n", prod);
}
```

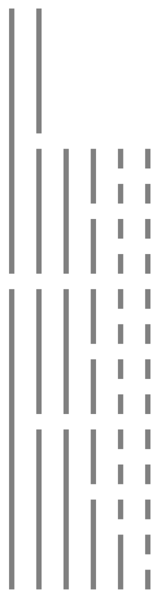


**Figure 5.2:** *The output of DDMIN on the program of Figure 5.1, and the re-execution of DDMIN.*

contained no call to it anymore (at line 15), the granularity had already reached line (*i.e.*, unit) level. However, *add* could only be removed as a whole, not line-by-line, as removing any single line would cause syntax errors. (This is one of the shortcomings of DDMIN that the grammar-based reducers wanted to fix.) So, DDMIN has produced a 1-minimal result (shown in Figure 5.2), but it is clearly not a global minimum. What we can realize when looking at this result is that we could re-execute DDMIN on this program with the same testing function as the first time and we may be able to remove the superfluous *add* function as well. Again, the gray bars on the right of the program code show the progress of DDMIN, and indeed, the subsets of the second configuration aligned well with the structure of this input and made further reduction possible. The result of the second execution of DDMIN is given in Figure 5.3. This is the global optimum for this example, so further executions of DDMIN are not visualized.

Motivated by this example, we can formalize the intuition that DDMIN could be executed multiple times. Since it cannot be told *a priori* how many executions are needed for a given input, we propose to iterate DDMIN until a fixed point is reached. We will denote the fixed-point iteration of DDMIN as

```
int mul(int a, int b)
{
    return a * b;
}
void main()
{
    int prod = 1;
    for (int i = 1; i <= 10; i++)
    {
        prod = mul(prod, i);
    }
    printf("prod: %d\n", prod);
}
```

**Figure 5.3:** *The output of DDMIN on the program of Figure 5.2.*

DDMIN* – following the notation used for HDD and HDD* [25] – and define it as follows:

$$ddmin^*(c_{\text{✗}}) = \begin{cases} c'_{\text{✗}} & \text{if } c_{\text{✗}} = c'_{\text{✗}} \\ ddmin^*(c'_{\text{✗}}) & \text{otherwise} \end{cases}$$
$$\text{where } c'_{\text{✗}} = ddmin(c_{\text{✗}}).$$

Although the asterisk notation is the same for the two algorithms and even its meaning is identical in both cases (*i.e.*, fixed-point iteration), its purpose is fundamentally different for HDD and DDMIN. A single execution of HDD has no minimality guarantees, only HDD* produces 1-tree-minimal results. However, even a single execution of DDMIN is guaranteed to give a 1-minimal result. The purpose of iterating it further is to find an even better 1-minimum. (Re-executing DDMIN does not guarantee better results in all cases, only if the configuration aligns well with the structure of the input.)

## 5.1 Evaluation

The experimental setup described in Chapter 3 was used to evaluate the effects of the proposed optimizations. Beyond those settings, there are some details that are specific to this chapter: DDMIN can be impractically slow on huge input configurations (*e.g.*, large input at character-level granularity), but fortunately, an extra option is available in Picire: a combined reduction pass that may achieve smaller outputs faster. The first pass splits the input into lines and uses them as a configuration. Line-based reduction is faster than character-based; however, it may produce larger results, as superfluous characters might be removed from the lines with finer granularity. The second pass then uses the output of the first, then continues the reduction at character-level granularity to achieve smaller results. We have not found a reference to this technique in the literature (and there is no theoretical guarantee that it will be faster for all inputs), however, it is really useful from a practical perspective, and therefore, we have used it in our experiments. Picire is implemented to maximally utilize each reduction pass: the line-based reduction continues until the fixed point is reached, then the character-based reduction does the same thing. Seemingly, this causes extra testing steps, but the experiments show that using this two-pass reduction is beneficial.

DDMIN may also give noticeably suboptimal results when the input has some well-defined structure over its units, which is quite typical for inputs for programs. During its iterations, DDMIN can separate the configuration in a way that is completely unaligned with the structure of the input, leading to incorrectly formatted, thus nonreproducible, and therefore, completely unusable intermediate test cases. Several studies have made efforts to address this problem, one of the most well-known approaches is HDD. However, if DDMIN* forces the reduction further with the help of the fixed-point iteration, it raises the question of *whether DDMIN* can compete with more sophisticated techniques (e.g., HDD*) in terms of effectiveness.* To examine this question, HDD* is included in the experiments using the Picireny project.

The experiments include two-pass DDMIN, DDMIN*, and the tree-based HDD* algorithm, therefore, a unified unit of measure must be used. Using the above-described non-whitespace characters as the “absolute” unit helps us to make a fair comparison between the different algorithms and their variants.

The workstation used to carry out the experiments was equipped with an Intel® Xeon® CPU E5-2680 v4 CPU clocked at 2.4 GHz and 128 GB RAM. The machine was running Ubuntu 22.04 with Linux kernel 5.15.0 and it was having no other load during the experiments.

## 5.1.1 Effectiveness of DDMIN*

### Character-Level Granularity

First, the character-level reduction has been evaluated on JRTS; Table 5.1 presents the results of this experiment. For each test case, the first group of values shows the properties of the inputs: the name and the size expressed in the unit of reduction. Then, the second group is the baseline data, *i.e.*, those measured using the traditional DDMIN algorithm: the number of testing steps needed to accomplish the reduction, and the size of the output. (The number of testing steps includes actual test case evaluations only; *i.e.*, does not include the re-evaluation of already seen configurations.) The last group of values contains the data collected during the executions of the fixed-point iterated algorithm (DDMIN*) on the test cases. In addition to the absolute numbers, we also give the changes relative to the baseline data. Plus, the number of iterations necessary to reach the fixed point is also shown.

**Table 5.1:** *Results with Character Granularity*

| Test Case | | DDMIN | | DDMIN* | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | Chars | Steps | Chars | Iters | Steps | | Chars | |
| jerry-3299 | 1,767 | 4,959 | 542 | 12 | 13,287 | *+167.94%* | 130 | *-76.01%* |
| jerry-3361 | 1,953 | 2,276 | 427 | 12 | 9,950 | *+337.17%* | 175 | *-59.02%* |
| jerry-3376 | 6,626 | 15,008 | 1,216 | 9 | 31,602 | *+110.57%* | 306 | *-74.84%* |
| jerry-3408 | 2,681 | 4,869 | 557 | 7 | 9,707 | *+99.36%* | 178 | *-68.04%* |
| jerry-3431 | 1,065 | 2,228 | 207 | 6 | 3,120 | *+40.04%* | 58 | *-71.98%* |
| jerry-3433 | 961 | 588 | 74 | 4 | 681 | *+15.82%* | 14 | *-81.08%* |
| jerry-3437 | 6,597 | 11,764 | 1,017 | 6 | 15,982 | *+35.86%* | 105 | *-89.68%* |
| jerry-3479 | 5,201 | 17,391 | 2,383 | 15 | 66,097 | *+280.06%* | 452 | *-81.03%* |
| jerry-3483 | 492 | 388 | 42 | 4 | 516 | *+32.99%* | 17 | *-59.52%* |
| jerry-3506 | 3,760 | 5,797 | 658 | 8 | 12,322 | *+112.56%* | 192 | *-70.82%* |
| jerry-3523 | 3,928 | 8,666 | 832 | 6 | 15,542 | *+79.34%* | 206 | *-75.24%* |
| jerry-3534 | 1,927 | 3,407 | 378 | 6 | 6,512 | *+91.14%* | 128 | *-66.14%* |
| jerry-3536 | 829 | 1,119 | 163 | 3 | 1,628 | *+45.49%* | 147 | *-9.82%* |

If an input is not optimal (*i.e.*, it could be reduced somehow), then at least two iterations of DDMIN* are expected: whenever an iteration manages to reduce the configuration, there will be a next iteration that tries to continue the reduction. If the configuration cannot be reduced further in an iteration, the algorithm halts. The iteration counts in Table 5.1 support this expectation. The highest number was 15 (for test case jerry-3479), and the lowest was 3 (also signaling that DDMIN* could always further reduce the result of DDMIN). Also, when the number of iterations is exactly two (not present in Table 5.1), that means that DDMIN* is not able to produce smaller results than DDMIN.

For all of the test cases, DDMIN* produced significantly smaller results than DDMIN. The output configuration got smaller by a minimum of 9.82% (in the case of jerry-3536) and by a maximum of 89.68% (for jerry-3437), and the average improvement was 67.94%. The cost of these improvements was an increased number of test executions, in the range of 15.82% and 337.17%, 111.41% being the average. Let us highlight the experiment of the jerry-3479 test case, which shows some impressive results. In that case, the input contained 5,201 characters, and DDMIN could reduce it to 2,383 characters, which is 45.81% of the input. However, the 15 iterations of DDMIN* could reduce it further to 433 characters, which is only 8.69% of the input size.

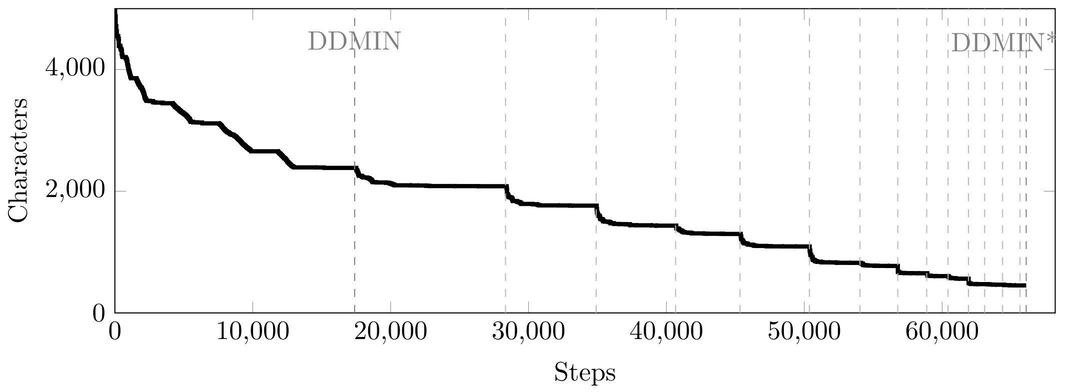


**Figure 5.4:** *The process of reduction of the jerry-3479 test case with DDMIN* using character granularity through 15 iterations: the change of configuration size over testing steps.*

Figure 5.4 presents the character-level reduction process of *jerry-3479* over 15 iterations. Changes in configuration size are represented along the vertical axis, from the input size to the end result of the last iteration. Test executions are represented along the horizontal axis, and each dashed vertical line represents the end of an iteration. The leftmost dashed line corresponds to the result of the first iteration, which is also the result of DDMIN (labeled DDMIN). The following iterations yield gradual reductions until the 15th iteration cannot reduce its input further; therefore, the iteration halts (rightmost dashed line labeled with DDMIN*). Bigger parts from the configuration could be removed close to the beginning of each iteration (as the splitting of the configuration is reset to 2), and then only smaller chunks could be removed as the process progresses (flat parts of the figure). Then again, larger removals can be observed at the beginning of each additional DDMIN* iteration, which is responsible for the improvement. The first iteration is responsible for 26.31% of the steps, then the second for 16.55%, and the following iterations gradually perform fewer steps.

### Line-Level Granularity

Table 5.2 presents the results of the line-level reduction of the test cases of JRTS and PTS. In addition to the previous table structure, we present the amount of time needed to perform these steps on the workstation used for the experiment, and the size is also expressed as the number of non-whitespace characters in the output.

The algorithm behavior in terms of efficiency is similar in both test suites. Not surprisingly, DDMIN* required more testing steps than DDMIN, the average increase is 66.08%, with 519.21% being the maximum (*gcc-61917*). Compared to this, the increase in wall clock time is only 28.59% with 280.25% being the maximum (also *gcc-61917*). For most of the test cases, DDMIN* produced smaller results than DDMIN. Since the configuration units are *lines*, the results should be compared in this unit. The output configuration got smaller by a maximum of 96.45% (*clang-27747*), and the average improvement is 48.08%. Interestingly, two different behaviors can be observed: DDMIN* improves the effectiveness of the reduction to a great extent on PTS (68.70%), which contains 86 times larger inputs (in terms of lines) than JRTS on average. However, the improvement is only observable in 5 of 13 cases of JRTS, and the average improvement is only 19.53%. After a manual examination of the output files, it turned out that where DDMIN* produced the same output as DDMIN, that was the global optimum and could not be reduced further at line-level. For smaller configurations (in our experiments, 18–118 units) DDMIN *might* give a global optimum; however, if the input configuration was bigger (36–20,617 units), DDMIN* could be very helpful in creating smaller outputs. An overlap can be observed between the intervals, where relatively small configurations could also be reduced further with DDMIN*: but eventually, the behavior of the algorithm is test case specific, and we could not unambiguously generalize the phenomenon.

**Table 5.2:** *Results with Line Granularity*

| Test Case | DDMIN | | | | DDMIN* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Steps | Time (s) | Lines | Chars | Steps | | Time (s) | | Lines | | Chars | |
| jerry-3299 | 54 | 4.33 | 13 | 307 | 65 | *+20.37%* | 5.24 | *+21.02%* | 13 | — | 307 | — |
| jerry-3361 | 28 | 3.87 | 4 | 165 | 29 | *+3.57%* | 3.88 | *+0.26%* | 4 | — | 165 | — |
| jerry-3376 | 144 | 15.22 | 17 | 361 | 181 | *+25.69%* | 18.14 | *+19.19%* | 5 | *-70.59%* | 113 | *-68.70%* |
| jerry-3408 | 22 | 1.74 | 3 | 165 | 22 | — | 1.81 | *+4.02%* | 3 | — | 165 | — |
| jerry-3431 | 29 | 3.24 | 5 | 104 | 35 | *+20.69%* | 3.89 | *+20.06%* | 3 | *-40.00%* | 56 | *-46.15%* |
| jerry-3433 | 23 | 3.15 | 2 | 57 | 23 | — | 3.29 | *+4.44%* | 2 | — | 57 | — |
| jerry-3437 | 186 | 20.27 | 20 | 497 | 245 | *+31.72%* | 24.93 | *+22.99%* | 10 | *-50.00%* | 252 | *-49.30%* |
| jerry-3479 | 148 | 16.72 | 15 | 515 | 186 | *+25.68%* | 19.30 | *+15.43%* | 10 | *-33.33%* | 364 | *-29.32%* |
| jerry-3483 | 13 | 1.90 | 2 | 56 | 13 | — | 1.90 | — | 2 | — | 56 | — |
| jerry-3506 | 33 | 4.21 | 3 | 93 | 34 | *+3.03%* | 4.33 | *+2.85%* | 3 | — | 93 | — |
| jerry-3523 | 49 | 6.39 | 4 | 90 | 51 | *+4.08%* | 6.67 | *+4.38%* | 4 | — | 90 | — |
| jerry-3534 | 85 | 8.95 | 10 | 223 | 95 | *+11.76%* | 10.17 | *+13.63%* | 4 | *-60.00%* | 121 | *-45.74%* |
| jerry-3536 | 33 | 3.22 | 7 | 158 | 38 | *+15.15%* | 3.66 | *+13.66%* | 7 | — | 158 | — |
| clang-22382 | 4,936 | 1,514.18 | 596 | 14,705 | 9,722 | *+96.96%* | 2,032.92 | *+34.26%* | 314 | *-47.32%* | 10,451 | *-28.93%* |
| clang-22704 | 11,930 | 5,957.04 | 919 | 9,572 | 16,695 | *+39.94%* | 6,541.71 | *+9.81%* | 129 | *-85.96%* | 1,748 | *-81.74%* |
| clang-23309 | 8,736 | 4,439.29 | 803 | 23,616 | 15,414 | *+76.44%* | 5,975.74 | *+34.61%* | 459 | *-42.84%* | 20,048 | *-15.11%* |
| clang-23353 | 6,117 | 2,127.34 | 670 | 11,337 | 12,243 | *+100.15%* | 2,769.17 | *+30.17%* | 236 | *-64.78%* | 5,147 | *-54.60%* |
| clang-25900 | 13,086 | 7,662.78 | 1,000 | 13,031 | 18,948 | *+44.80%* | 8,496.11 | *+10.88%* | 201 | *-79.90%* | 5,783 | *-55.62%* |
| clang-26350 | 21,027 | 20,445.25 | 1,485 | 63,740 | 34,880 | *+65.88%* | 23,424.97 | *+14.57%* | 335 | *-77.44%* | 16,673 | *-73.84%* |
| clang-26760 | 35,647 | 25,824.05 | 1,782 | 17,170 | 42,565 | *+19.41%* | 22,886.70 | *-11.37%* | 209 | *-88.27%* | 6,755 | *-60.66%* |
| clang-27747 | 17,309 | 15,970.98 | 1,407 | 16,825 | 22,208 | *+28.30%* | 16,837.55 | *+5.43%* | 50 | *-96.45%* | 1,905 | *-88.68%* |
| clang-31259 | 9,172 | 5,510.85 | 764 | 15,395 | 17,421 | *+89.94%* | 6,951.33 | *+26.14%* | 245 | *-67.93%* | 9,742 | *-36.72%* |
| gcc-59903 | 8,677 | 7,453.63 | 805 | 15,900 | 14,440 | *+66.42%* | 8,504.36 | *+14.10%* | 459 | *-42.98%* | 13,823 | *-13.06%* |
| gcc-61383 | 21,857 | 7,391.74 | 2,847 | 45,370 | 99,322 | *+354.42%* | 22,975.37 | *+210.82%* | 1,207 | *-57.60%* | 16,685 | *-63.22%* |
| gcc-61917 | 51,876 | 26,518.21 | 6,975 | 92,919 | 321,223 | *+519.21%* | 100,834.93 | *+280.25%* | 2,251 | *-67.73%* | 27,094 | *-70.84%* |
| gcc-64990 | 14,500 | 15,754.00 | 1,173 | 21,002 | 23,157 | *+59.70%* | 17,466.87 | *+10.87%* | 447 | *-61.89%* | 17,528 | *-16.54%* |
| gcc-65383 | 8,536 | 4,106.35 | 715 | 16,034 | 14,093 | *+65.10%* | 4,932.27 | *+20.11%* | 273 | *-61.82%* | 11,297 | *-29.54%* |
| gcc-66186 | 9,326 | 9,297.62 | 803 | 17,442 | 17,334 | *+85.87%* | 10,373.58 | *+11.57%* | 328 | *-59.15%* | 11,837 | *-32.14%* |
| gcc-66375 | 11,352 | 11,476.84 | 1,062 | 18,363 | 19,958 | *+75.81%* | 13,354.05 | *+16.36%* | 327 | *-69.21%* | 11,219 | *-38.90%* |
| gcc-70127 | 20,547 | 17,735.92 | 1,620 | 21,367 | 35,790 | *+74.19%* | 19,839.98 | *+11.86%* | 337 | *-79.20%* | 9,183 | *-57.02%* |
| gcc-71626 | 1,437 | 161.77 | 130 | 4,652 | 1,784 | *+24.15%* | 184.27 | *+13.91%* | 18 | *-86.15%* | 611 | *-86.87%* |

## Two-Pass Reduction

Not only the manual test case minimization can be time-consuming when the input is huge, but DDMIN can also take a lot of time. Although line-level reduction had reasonable time requirements, the character-level reduction was completely unacceptable for practical use in the case of PTS. However, the line-level reduction does not exploit the full potential of the algorithm, and there is an intermediate way, which might be "fast enough" and the output is still smaller. (*E.g.,* Picire reduced the *gcc-71626* benchmark program to 4,652 non-whitespace characters with DDMIN at line-level in 1,437 steps, furthermore, at character-level, it could be reduced to 8,713 non-whitespace characters in *121,968* steps. It was using more resources and the results it gave were even worse, which is not the best combination.) Therefore, a combined reduction pass has been utilized that first reduces the input with line granularity, and then reduces further with character-level granularity. The reduction time became more acceptable using this technique as the line-level granularity.

Table 5.3 shows the results of this two-pass reduction with DDMIN and DDMIN*. The structure is similar to Table 5.2, only the main basis of comparison is changed to non-whitespace characters (column "Chars") to avoid misinterpretation of the results. The average increase in the testing steps is 96.41%, with 615.33% being the maximum (*clang-23309*), which means that a maximum of seven times more testing steps are needed to reduce tests from our suite. The increase in wall clock time is 68.91% with 711.25% being the maximum (*clang-23309*). Compared to the line-level experiment, DDMIN* produced smaller results in 8 of 13 (+3) on JRTS, meaning the line-level global minimum could be reduced further with finer granularity. The output got smaller by a maximum of 88.27% (*clang-27747*), and the average improvement was 45.76%.

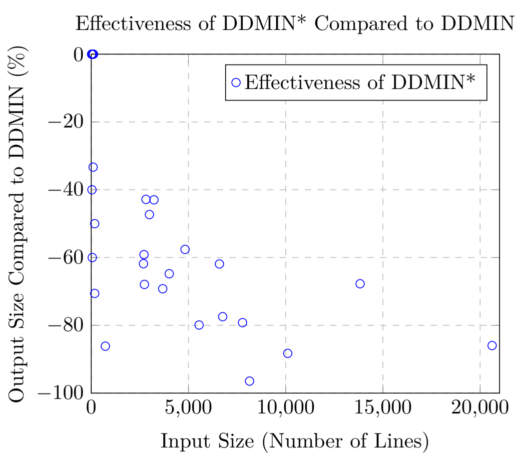
Based on the two-pass reduction, it is definitely worth using DDMIN* if the goal is to produce as small outputs as possible with the least information about the input structure (*i.e.*, lines and characters only) since the size of the output halved on average. Results of Table 5.3 can be interpreted in a way that results from Table 5.2 were given to Picire to further reduce it with character-level DDMIN*. Comparing the results from the two tables, the average improvement is *53.96%*, but it has its price in the increased time needed to accomplish the reduction: 15 times more wall-clock time is needed on average. (This might sound like a lot, however, the slowest reduction in JRTS needed 6 minutes, which can be made even more acceptable with parallel execution in a fuzzer ecosystem.)

**Table 5.3:** *Results with Combined Granularity*

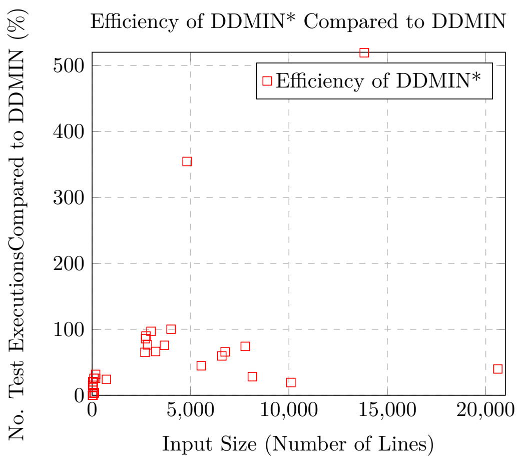
| Test Case | DDMIN | | | DDMIN* | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Steps | Time (s) | Chars | Steps | | Time (s) | | Chars | |
| jerry-3299 | 1,092 | 81.94 | 143 | 2,209 | *+102.29%* | 165.22 | *+101.64%* | 118 | *-17.48%* |
| jerry-3361 | 625 | 54.56 | 112 | 1,181 | *+88.96%* | 95.33 | *+74.73%* | 89 | *-20.54%* |
| jerry-3376 | 1,549 | 137.97 | 201 | 993 | *-35.89%* | 82.94 | *-39.89%* | 89 | *-55.72%* |
| jerry-3408 | 423 | 31.97 | 63 | 575 | *+35.93%* | 42.12 | *+31.75%* | 58 | *-7.94%* |
| jerry-3431 | 456 | 44.45 | 59 | 358 | *-21.49%* | 32.61 | *-26.64%* | 31 | *-47.46%* |
| jerry-3433 | 153 | 16.57 | 21 | 172 | *+12.42%* | 17.37 | *+4.83%* | 21 | — |
| jerry-3437 | 1,837 | 167.81 | 166 | 1,189 | *-35.27%* | 108.42 | *-35.39%* | 107 | *-35.54%* |
| jerry-3479 | 1,875 | 162.08 | 253 | 2,124 | *+13.28%* | 174.25 | *+7.51%* | 191 | *-24.51%* |
| jerry-3483 | 40 | 5.30 | 4 | 42 | *+5.00%* | 5.20 | *-1.89%* | 4 | — |
| jerry-3506 | 282 | 328.47 | 48 | 329 | *+16.67%* | 386.24 | *+17.59%* | 48 | — |
| jerry-3523 | 377 | 35.52 | 63 | 440 | *+16.71%* | 39.69 | *+11.74%* | 63 | — |
| jerry-3534 | 1,079 | 92.67 | 148 | 1,037 | *-3.89%* | 84.73 | *+-8.57%* | 105 | *-29.05%* |
| jerry-3536 | 683 | 56.08 | 148 | 842 | *+23.28%* | 67.73 | *+20.77%* | 148 | — |
| clang-22382 | 97,305 | 7,018 | 7,431 | 312,664 | *+221.32%* | 64,387.15 | *+153.26%* | 2,700 | *-63.67%* |
| clang-22704 | 48,929 | 5,493 | 4,365 | 54,819 | *+12.04%* | 10,341.69 | *-19.90%* | 694 | *-84.10%* |
| clang-23309 | 128,663 | 9,971 | 11,736 | 920,361 | *+615.33%* | 268,369.71 | *+711.25%* | 5,341 | *-54.49%* |
| clang-23353 | 69,955 | 5,041 | 7,352 | 204,809 | *+192.77%* | 35,926.18 | *+144.34%* | 2,612 | *-64.47%* |
| clang-25900 | 107,132 | 10,842 | 7,647 | 271,594 | *+153.51%* | 57,745.12 | *+66.18%* | 2,574 | *-66.34%* |
| clang-26350 | 571,649 | 66,923 | 47,655 | 1,031,666 | *+80.47%* | 275,089.75 | *-11.17%* | 6,646 | *-86.05%* |
| clang-26760 | 175,770 | 19,461 | 9,456 | 205,756 | *+17.06%* | 54,985.43 | *-4.77%* | 2,551 | *-73.02%* |
| clang-27747 | 101,777 | 14,389 | 8,988 | 71,072 | *-30.17%* | 23,191.19 | *-42.01%* | 1,054 | *-88.27%* |
| clang-31259 | 97,311 | 10,840 | 8,156 | 215,736 | *+121.70%* | 54,647.29 | *+83.45%* | 4,250 | *-47.89%* |
| gcc-59903 | 108,168 | 11,997 | 10,080 | 601,419 | *+456.00%* | 145,742.81 | *+355.33%* | 5,803 | *-42.43%* |
| gcc-61383 | 374,540 | 49,046 | 32,544 | 801,846 | *+114.09%* | 243,083.31 | *+34.40%* | 9,817 | *-69.83%* |
| gcc-61917 | 1,025,065 | 174,767 | 65,357 | 1,685,350 | *+64.41%* | 576,466.30 | *-20.89%* | 11,268 | *-82.76%* |
| gcc-64990 | 294,382 | 16,967 | 10,051 | 791,075 | *+168.72%* | 224,668.02 | *+145.05%* | 5,042 | *-49.84%* |
| gcc-65383 | 106,058 | 9,438 | 7,593 | 438,661 | *+313.60%* | 105,516.61 | *+272.20%* | 3,805 | *-49.89%* |
| gcc-66186 | 95,930 | 11,196 | 9,236 | 254,214 | *+165.00%* | 76,236.37 | *+124.48%* | 5,144 | *-44.30%* |
| gcc-66375 | 110,042 | 14,186 | 9,680 | 194,366 | *+76.63%* | 51,084.57 | *+35.09%* | 4,932 | *-49.05%* |
| gcc-70127 | 149,023 | 21,531 | 12,062 | 272,148 | *+82.62%* | 62,918.22 | *+11.66%* | 2,498 | *-79.29%* |
| gcc-71626 | 17,879 | 1,172 | 1,644 | 8,145 | *-54.44%* | 568.14 | *-59.84%* | 252 | *-84.67%* |

### 5.1.2 Input-Dependent Behavior of Algorithm Variants

Figure 5.5 visualizes the raw data from Table 5.2. Each mark on the charts represents a test case reduced with DDMIN*, and the position of the mark reflects how the fixed-point iteration affected the reduction compared to DDMIN. The input configuration size is represented along the horizontal axis while the effects of DDMIN* are represented along the vertical axis (all relative to the size and number of testing steps of the baseline). Figure 5.5(a) corresponds to the effectiveness, it shows that DDMIN* produced exactly the same output as DDMIN in some cases, however, the outputs are a fraction of the baseline for



**(a)** *Effectiveness of DDMIN* Compared to DDMIN.*

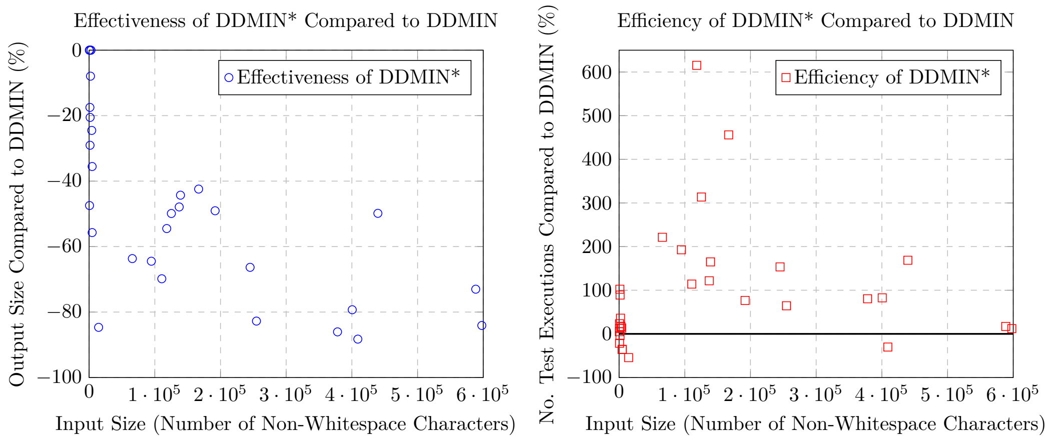


**(b)** *Efficiency of DDMIN* Compared to DDMIN.*

**Figure 5.5:** *Effect of line-level DDMIN* as a Function of the* Size *of the Input.*

the majority of benchmarks. Figure 5.5(b) shows the effect on the efficiency of the reduction. DDMIN* yielded the smaller outputs slower (as expected), the additional cost in the number of testing steps is not expensive for inputs with configuration size less than 1000 (*13.28%*), and the effort is doubled for larger inputs (*109.56%* on average including the outliers, and *66.08%* when they are excluded). There are only a handful of outliers where DDMIN* required much more testing steps than the baseline, namely *gcc-61383* and *gcc-61917*.

Figure 5.6 visualizes the raw data from Table 5.3. Its structure is similar to Figure 5.5, the only difference is the metrics of the configuration size represented along the horizontal axis: the number of lines has been replaced by the number of non-whitespace characters. Figure 5.6(a) shows similar results as Figure 5.5(a), DDMIN* produces smaller outputs for the vast majority of benchmarks, and some small inputs cannot be reduced further with fixed-point iteration. Figure 5.6(b) shows surprising results: some inputs could be reduced faster. The reason behind the efficiency improvement is that Picire can produce smaller results at line level in a reasonable amount of testing steps with DDMIN*, then the character-level reduction can work further starting from this smaller input configuration. However, the general case is that DDMIN* requires more testing steps, furthermore, unlike the line-level reduction, the in-



**(a)** *Effectiveness of DDMIN* Compared to DDMIN.*

**(b)** *Efficiency of DDMIN* Compared to DDMIN.*

**Figure 5.6:** *Effect of two-pass DDMIN* as a Function of the* Size *of the Input.*

crease did not only double but quadrupled on average with two-pass reduction. There are two different outliers where DDMIN* required many more testing steps, namely *clang-23309* and *gcc-59903*.

### 5.1.3 Relation to More Sophisticated Techniques

Seeing these promising results, the question can be raised whether DDMIN* can compete with HDD* in terms of output size. There is no doubt that neither DDMIN nor DDMIN* can compete with HDD* in terms of the number of required testing steps. HDD takes the input structure into account, which makes it highly efficient compared to structure-unaware algorithms. The answer can be found in Table 5.4. While the output of traditional DDMIN is 720% larger than the output of HDD* on average (in these test suites), DDMIN* brings the results much closer. The output of DDMIN* is 139% larger than the output of HDD*, which is not that bad for a structure-unaware technique. However, DDMIN* is still far away from being a competitor to HDD* and should be optimized further.

In the tables below, characters serve as an absolute measure (column “Chars”), and it is expressed as the number of non-whitespace characters to avoid bias from indentation or other formatting differences.

**Table 5.4:** *Results with HDD* and Combined Granularity DDMIN**

| Test Case | HDD* | | | DDMIN* | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Steps | Time (s) | Chars | Steps | | Time (s) | | Chars | |
| jerry-3299 | 171 | 17.80 | 92 | 2,209 | *+1,191.81%* | 165.22 | *+828.20%* | 118 | *+28.26%* |
| jerry-3361 | 141 | 12.40 | 97 | 1,181 | *+737.59%* | 95.33 | *+668.79%* | 89 | *-8.25%* |
| jerry-3376 | 116 | 11.71 | 70 | 993 | *+756.03%* | 82.94 | *+608.28%* | 89 | *+27.14%* |
| jerry-3408 | 164 | 12.00 | 62 | 575 | *+250.61%* | 42.12 | *+251.00%* | 58 | *-6.45%* |
| jerry-3431 | 52 | 5.44 | 31 | 358 | *+588.46%* | 32.61 | *+499.45%* | 31 | — |
| jerry-3433 | 10 | 1.87 | 21 | 172 | *+1,620.00%* | 17.37 | *+828.88%* | 21 | — |
| jerry-3437 | 38 | 5.43 | 42 | 1,189 | *+3,028.95%* | 108.42 | *+1,896.69%* | 107 | *+154.76%* |
| jerry-3479 | 225 | 22.56 | 120 | 2,124 | *+844.00%* | 174.25 | *+672.38%* | 191 | *+59.17%* |
| jerry-3483 | 67 | 6.29 | 38 | 42 | *-37.31%* | 5.20 | *-17.33%* | 4 | *-89.47%* |
| jerry-3506 | 112 | 10.76 | 52 | 329 | *+193.75%* | 386.24 | *+3,489.59%* | 48 | *-7.69%* |
| jerry-3523 | 109 | 9.72 | 63 | 440 | *+303.67%* | 39.69 | *+308.33%* | 63 | — |
| jerry-3534 | 170 | 14.32 | 96 | 1,037 | *+510.00%* | 84.73 | *+491.69%* | 105 | *+9.38%* |
| jerry-3536 | 146 | 11.95 | 123 | 842 | *+476.71%* | 67.73 | *+466.78%* | 148 | *+20.33%* |
| clang-22382 | 14,842 | 4,618.19 | 582 | 312,664 | *+2,006.62%* | 64,387.15 | *+1,294.21%* | 2,700 | *+363.92%* |
| clang-22704 | 10,530 | 11,665.69 | 168 | 54,819 | *+420.60%* | 10,341.69 | *-11.35%* | 694 | *+313.10%* |
| clang-23309 | 24,594 | 22,821.60 | 3,582 | 920,361 | *+3,642.22%* | 268,369.71 | *+1,075.95%* | 5,341 | *+49.11%* |
| clang-23353 | 14,585 | 5,739.18 | 374 | 204,809 | *+1,304.24%* | 35,926.18 | *+525.98%* | 2,612 | *+598.40%* |
| clang-25900 | 14,737 | 8,753.28 | 1,562 | 271,594 | *+1,742.94%* | 57,745.12 | *+559.70%* | 2,574 | *+64.79%* |
| clang-26350 | 16,748 | 21,945.57 | 1,613 | 1,031,666 | *+6,059.94%* | 275,089.75 | *+1,153.51%* | 6,646 | *+312.03%* |
| clang-26760 | 12,925 | 13,059.59 | 586 | 205,756 | *+1,491.92%* | 54,985.43 | *+321.03%* | 2,551 | *+335.32%* |
| clang-27747 | 7,164 | 5,147.64 | 419 | 71,072 | *+892.07%* | 23,191.19 | *+350.52%* | 1,054 | *+151.55%* |
| clang-31259 | 18,900 | 19,467.98 | 2,174 | 215,736 | *+1,041.46%* | 54,647.29 | *+180.70%* | 4,250 | *+95.49%* |
| gcc-59903 | 18,646 | 18,169.24 | 1,726 | 601,419 | *+3,125.46%* | 145,742.81 | *+702.14%* | 5,803 | *+236.21%* |
| gcc-61383 | 17,279 | 13,640.33 | 1,704 | 801,846 | *+4,540.58%* | 243,083.31 | *+1,682.09%* | 9,817 | *+476.12%* |
| gcc-61917 | 17,276 | 12,082.91 | 1,764 | 1,685,350 | *+9,655.44%* | 576,466.30 | *+4,670.92%* | 11,268 | *+538.78%* |
| gcc-64990 | 19,258 | 27,572.61 | 2,866 | 791,075 | *+4,007.77%* | 224,668.02 | *+714.82%* | 5,042 | *+75.92%* |
| gcc-65383 | 11,836 | 8,757.59 | 1,028 | 438,661 | *+3,606.16%* | 105,516.61 | *+1,104.86%* | 3,805 | *+270.14%* |
| gcc-66186 | 15,649 | 18,205.31 | 2,617 | 254,214 | *+1,524.47%* | 76,236.37 | *+318.76%* | 5,144 | *+96.56%* |
| gcc-66375 | 21,171 | 35,216.11 | 2,963 | 194,366 | *+818.08%* | 51,084.57 | *+45.06%* | 4,932 | *+66.45%* |
| gcc-70127 | 21,562 | 46,974.24 | 1,763 | 272,148 | *+1,162.16%* | 62,918.22 | *+33.94%* | 2,498 | *+41.69%* |
| gcc-71626 | 4,210 | 454.84 | 168 | 8,145 | *+93.47%* | 568.14 | *+24.91%* | 252 | *+50.00%* |

## 5.2 Conclusions

We have evaluated the fixed-point iteration of minimizing Delta Debugging (DDMIN*) in two, slightly different settings. The test suites used are publicly available and have already been used in reduction-related studies. First, the reduction of test cases was performed with character level granularity on JRTS; the output became smaller by 67.94% on average. Then, reduction with line granularity was performed, and the experiments show that DDMIN* can produce 48.08% smaller outputs on average (68.70% on Perses Test Suite and 19.53% on JerryScript Reduction Test Suite). The price of this improvement is the increased number of steps, which was 66.08% on average. Then, a “combined”, two-pass reduction was performed where test cases were first reduced with line granularity, then these intermediate results were reduced further with character granularity as fine-tuning. DDMIN* overcame DDMIN with this setting as well, and could reduce inputs further by 45.76% on average. Surprisingly some inputs could be reduced faster with DDMIN* as the line-level reduction produces results in a reasonable amount of steps, then the character-level reduction can work further from this smaller input configuration.

If grammar is not available or maintaining it is not a beneficial option, we would recommend using the combined, two-pass DDMIN*, since it was shown that the fixed-point iteration results in smaller outputs in exchange for some extra CPU cycles.

Encouraged by the promising results, we have compared the output of DDMIN* to the output of HDD*, to see whether a structure-unaware algorithm can compete with a “more clever” one. In terms of required testing steps, the answer is simply no; however, in terms of size DDMIN* brought the results much closer to each other, from 9 times larger outputs (DDMIN) to only 3 times larger ones. There is still room for improvement in those situations where the structure information is missing or changing rapidly.

Based on the experimental data and observations above, we can conclude the contributions of this chapter:

1. DDMIN* is most effective with character-level granularity (67.94% smaller outputs) compared to line-level reduction (48.08% smaller outputs). However, character-level reduction can be unacceptably slow for “large” inputs and line-level reduction leaves unnecessary parts in its output; therefore, we used a combined approach, where DDMIN* produced 45.76% smaller outputs. Furthermore, the two-pass reduction produced 53.96% smaller outputs than the line-level approach in our experiments, on average.

2. DDMIN* incurs an additional cost (number of testing steps), which appears in most cases. (Some tests from our experimental setup could be reduced with fewer steps, but these are exceptions.) This additional cost is related to the size of the test case, but it does not grow beyond all limits. We identified that a maximum of seven times more testing steps are needed with DDMIN* in the used benchmark suites. The effectiveness of the reduction shows similar patterns: the larger the input configuration, the larger the potential to reduce. There were some small test cases (JRTS) where DDMIN* could not reduce the input further, however, this cannot be completely generalized.

If the input configuration has some superfluous items, DDMIN* can reduce it further regardless of its size.

3. Encouraged by the promising results, we have compared the output of DDMIN* to the output of HDD*, whether a structure-unaware algorithm can compete with a “more clever”. In terms of required testing steps, the answer is simply no; however, DDMIN* brought the results much closer to each other, from 9 times larger outputs (DDMIN) to only 3 times larger ones.

*"Focus on long-term success, but be willing to make short-term adjustments to get there."*

— Simon Sinek

# 6 Parallel Optimizations of DDMIN*

Minimizing Delta Debugging is already more than twenty years old and still widely used because it works on any kind of input. Many approaches have tried to work smarter since the first appearance of DDMIN: HDD [25], Pardis [11], ReduKtor [29], *etc.* could all produce smaller output faster than DDMIN, but they typically needed some extra information about the structure of the test case, usually a grammar. This additional requirement can act as a blocker for some users of test case reducers: grammar may not be readily available, and writing (or maintaining) one may not be a practical option. In such cases, the structure-unaware nature of DDMIN is proven to be very useful.

This is why we have investigated whether it was possible to make DDMIN itself work faster without compromising its minimality guarantees. One technique that has already been proven useful for speeding up DDMIN is parallelization [18]. The question we sought the answer to was whether it was possible to make the *parallel DDMIN* even faster without losing the 1-minimal property of its output.

Hodován *et al.* [18] noticed that the original implementation of DDMIN uses sequential loops to realize the "reduce to subset" and "reduce to complement" phases, however, the potential for parallelization is present in Figure 2.1, *i.e.*, $\exists i \in \{1, \ldots, n\}$ does not specify how to find $i$. Since $n$ can be big for real

inputs (and testing a configuration is considered to be an expensive part of the algorithm), they rewrote DDMIN to use parallel loops. As testing different configurations is independent, their proposal worked well in practice and achieved 75-80% less runtime in their experiments. Figure 6.1(a) shows how sequential loops iterate through five configurations: if we assume that every *test* takes the same amount of time $t$, then checking all of them takes $5t$. However, if the loops are implemented in a parallel way, as shown in Figure 6.1(b), checking the configurations takes only $t$, which might bring a significant speedup to reduction. For the formal definition of their parallel DDMIN formulation, the reader is referred to [18]. Three assumptions were made regarding correctness and effectiveness, of which the following one is relevant to this study: When a *fail* is found in a parallel loop, the other active loop bodies should be aborted even if their computation has not finished yet. This might cause computation results to be thrown away, but it does not harm the minimality guarantees of the algorithm.
The key contributions of this part:

- describe the stability issues of the parallel DDMIN algorithm,
- provide a stabilization approach, and
- define an algorithm variant, named *GreeDDy*, that can speed up the parallel reduction.

Let $j$ be the parallelization capabilities of the algorithm, *i.e.*, how many $test(\Delta_i)$ or $test(\nabla_i)$ jobs can be started concurrently (five were used in Figure 6.1). Let $T$ be the *testing window* ($|T| = j$), *i.e.*, $j$ tests are executed and $j$ results (✓, ✗, or **?**) are produced. Let $F$ denote the set of configurations with a *fail* outcome in $T$; if $|F| > 1$ then the behavior of parallel DDMIN [18] becomes unstable: it will choose among the interesting configurations based on which produced its *fail* outcome first. Therefore, different test reductions can yield different outcomes, which is not appropriate for carrying out reproducible experiments. (Note that the 1-minimality of the algorithm is not harmed, since multiple local minima might exist.)

The “reduce to subset” and “reduce to complement” phases iterate through configurations in a forward or backward syntactic order, therefore, it is known which configuration *should* be investigated first. To stabilize the algorithm, the following changes must be made: if a *fail* is found in $T$, then no new parallel loop bodies are started (no change), and the active *test* executions should be

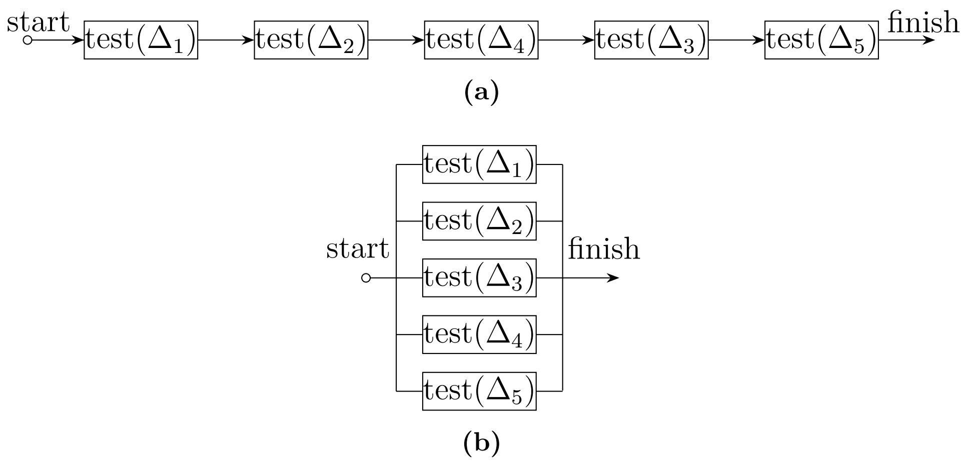


**Figure 6.1:** *(a) Sequential execution of "reduce to subset". (b) Parallel execution of "reduce to subset".*

awaited (*i.e.*, computation results are not thrown away). If multiple *fail*s are found, the syntactic order must be taken into account when choosing which one to reduce further.

When multiple *fail*s are found in $T$ and the algorithm chooses one of them based on the iterator, the results from other configurations are thrown away, even if they could have been useful. This results in superfluous test executions on configurations that have already been tested (and *fail*ed). The following strategy can help minimize the number of test executions: If a testing window has multiple *fail*s, then it is worth trying to combine those configurations that yielded them and check whether this combination also results in a *fail*. If yes, several test executions are saved in one step. If no, then select the first *fail* (based on the syntactic order) and try to combine the other interesting configurations one by one. This case can also save testing steps as only configurations with a *fail* outcomes are retested instead of the whole testing window in the next parallel loop iteration. Figure 6.2 formalizes our proposed optimization using the notations already discussed.

$$greeDDy(c'_{\text{✗}}) = greeDDy_2(c'_{\text{✗}}, 2) \text{ where}$$

$$greeDDy_2(c'_{\text{✗}}, n) = \begin{cases} greeDDy_2(\bigcap_{i \in F} \overline{C}_i, max(n - |F|, 2)) & \text{if } |F| > 1 \wedge test(\bigcap_{i \in F} \overline{C}_i) = \text{✗} \text{ (“reduce greedily”)} \\ greeDDy_2(\overline{C}_i, 2) & \text{else if } \exists i \in F \cdot \overline{C}_i \in \{\Delta_1, ..., \Delta_n\} \text{ (“reduce to subset”)} \\ greeDDy_2(\overline{C}_i, max(n-1, 2)) & \text{else if } \exists i \in F \cdot \overline{C}_i \in \{\nabla_1, ..., \nabla_n\} \text{ (“reduce to complement”)} \\ greeDDy_2(c'_{\text{✗}}, min(|c'_{\text{✗}}|, 2n)) & \text{else if } n < |c'_{\text{✗}}| \text{ (“increase granularity”)} \\ c'_{\text{✗}} & \text{otherwise (“done”).} \end{cases}$$

where $\overline{C}$ is a sequence of $C$, such that $\{\nabla_1, ..., \nabla_n\} \subseteq C \subseteq \{\Delta_1, ..., \Delta_n, \nabla_1, ..., \nabla_n\}$,
$\nabla_i = c'_{\text{✗}} - \Delta_i$, $c'_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all $\Delta_i$ are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✗}}|/n$ holds.
$F$ is a set of indices over $\overline{C}$ such that $\forall i \in F \cdot test(\overline{C}) = \text{✗}$ and $F = \emptyset \iff \nexists \overline{C_i} \cdot test(\overline{C_i}) = \text{✗}$.

**Figure 6.2:** *GreeDDy: the greedy extension of minimizing Delta Debugging.*

## 6.1 Evaluation

The experimental setup described in Chapter 3 was used to evaluate the effects of the proposed optimizations. Beyond those settings, there are some details that are specific to this chapter: the resource-efficient content-based caching was enabled (Section 4), and the testing window was chosen to be 4 ($|T| = j = 4$). The fixed-point iteration variant of DDMIN (DDMIN*) was used as a baseline. To see how our proposed extension performs on real-life test cases, we have taken 10 GCC-related test cases from the Perses Test Suite and minimized them with both DDMIN* and *GreeDDy** (the asterisk denotes that *GreeDDy* was iterated to a fixed point).

The workstation used to carry out the experiments was equipped with an Intel® Xeon® CPU E5-2680 v4 CPU clocked at 2.4 GHz and 128 GB RAM. The machine was running Ubuntu 22.04 with Linux kernel 5.15.0 and it was having no other load during the experiments.

Table 6.1 presents the results of reducing the test cases with lines as the unit of reduction. For each test case, the first group of values shows the properties of the inputs: the name and the size expressed in non-whitespace characters (column “Chars”). Then, the second group is the baseline data, *i.e.*, those measured using DDMIN*: the size of the output, the number of testing steps, and the runtime needed to accomplish the reduction. The last group of values contains the data collected during the executions of the greedy variant (*GreeDDy**) on the test cases. In addition to the absolute numbers, we also provide the changes relative to the baseline data.

For 6 of 10 test cases, *GreeDDy** produced different outputs than DDMIN*;

**Table 6.1:** *Results with Line Granularity*

| **Test Case** | | **DDMIN*** | | | ***GreeDDy**** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Name** | **Chars** | **Chars** | **Steps** | **Runtime (s)** | **Chars** | | **Steps** | | **Runtime (s)** | |
| gcc-61383 | 110,643 | 16,685 | 130,498 | 8,871.45 | 15,386 | *(-7.79%)* | 122,505 | *(-6.12%)* | 7,340.30 | *(-17.26%)* |
| gcc-64990 | 439,587 | 17,528 | 48,117 | 15,419.29 | 16,871 | *(-3.75%)* | 29,532 | *(-38.62%)* | 8,540.72 | *(-44.61%)* |
| gcc-65383 | 125,221 | 11,297 | 27,720 | 4,242.85 | 11,279 | *(-0.16%)* | 19,390 | *(-30.05%)* | 2,448.62 | *(-42.29%)* |
| gcc-66186 | 139,087 | 11,837 | 34,337 | 9,902.93 | 11,837 | *(—)* | 21,247 | *(-38.12%)* | 5,177.49 | *(-47.72%)* |
| gcc-66375 | 191,827 | 11,219 | 38,523 | 11,667.02 | 11,126 | *(-0.83%)* | 23,429 | *(-39.18%)* | 6,153.44 | *(-47.26%)* |
| gcc-66412 | 209,886 | 7,004 | 43,605 | 4,756.91 | 7,004 | *(—)* | 30,313 | *(-30.48%)* | 2,800.96 | *(-41.12%)* |
| gcc-66691 | 56,682 | 26,412 | 28,789 | 1,747.56 | 26,412 | *(—)* | 21,361 | *(-25.80%)* | 1,242.39 | *(-28.91%)* |
| gcc-70127 | 400,556 | 9,183 | 72,079 | 17,752.48 | 8,562 | *(-6.76%)* | 54,096 | *(-24.95%)* | 10,185.00 | *(-42.63%)* |
| gcc-70586 | 589,017 | 18,166 | 83,302 | 33,545.04 | 17,791 | *(-2.06%)* | 57,150 | *(-31.39%)* | 17,878.05 | *(-46.70%)* |
| gcc-71626 | 14,465 | 611 | 5,725 | 187.62 | 611 | *(—)* | 3,314 | *(-42.11%)* | 112.08 | *(-40.26%)* |
| **Average** | | | | | | ***-2.13%*** | | ***-30.68%*** | | ***-39.88%*** |

it got a bit smaller, and the average improvement was 2.13% (including those that produced exactly the same output as well). Supporting the motivation behind this research, the number of started test executions and the runtime were affected heavily. Greedily combining *fail*ed test executions required 30.68% fewer testing steps on average (42.11% fewer tests were needed in the best case and 6.12% of testing steps could be saved in the worst case). The runtime shows similar traits: On average, 39.88% of the execution time could be saved with *GreeDDy**; 47.42% in the best and 17.26% in the worst cases. According to these results, the started tests and the runtime properties of the algorithm show promising results, and based on them, it seems worthwhile to deal with an algorithm that is more than 20 years old.

## 6.2 Conclusions

Based on the experimental data and observations above, we can conclude the contributions of this chapter:

1. Investigated the stability issues of DDMIN* and provided a stabilization approach for it.

2. Made the parallel execution of DDMIN* even faster, exploiting the potentials of the already-seen testing windows (named *GreeDDy**).

3. Presented our idea where multiple test executions with *fail* outcomes could be merged to save further retesting, and found that *GreeDDy** could save 30.68% of the testing steps of DDMIN* which resulted in 39.88% less runtime. The output of *GreeDDy** usually got smaller, however, the effects of the algorithm on the output are negligible.

*“Creativity is the art of finding a new route to a known destination.”*

— Simon Sinek

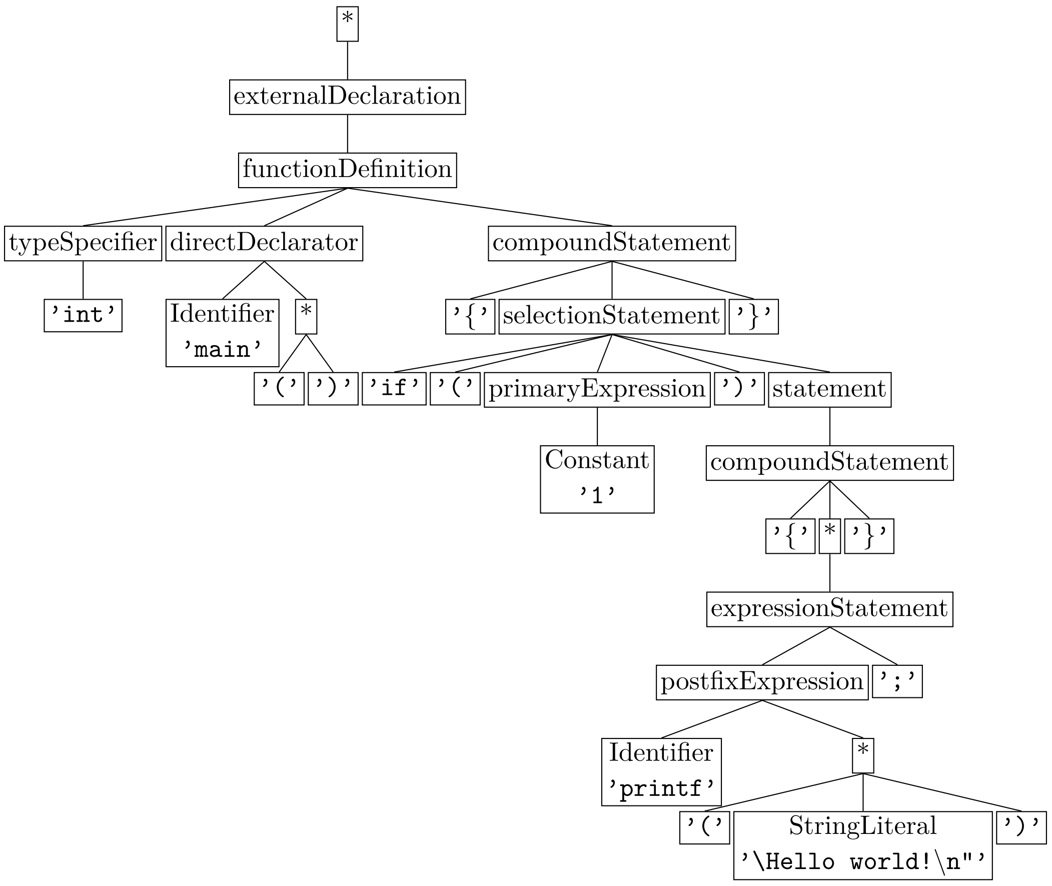
# 7 Extending Hierarchical Delta Debugging with Hoisting

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 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 [20], or by rotating recursive structures of the tree to reduce its height [19]. However, these transformations do not change the core structure of the tree, *i.e.*, the test case serialized from the preprocessed tree will still be the same as the original input. Because of this, and because of how the HDD variants operate, an HDD-reduced test case (even if 1-tree-minimal) may contain structural elements that a human expert would still remove.

A simple example of this suboptimal structure-preserving behavior is shown in Figure 7.1. The C program in Figure 7.1(a) prints the classic “Hello world!” message, and 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. (This is an example where the interesting property of the test case is *not* a program failure.) Figure 7.1(b) shows the parse tree of the program,

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

**(a)** *A "Hello World" program in C.*



**(b)** *Parse tree of the "Hello World" program.*

**Figure 7.1:** *An overly complicated "Hello World" program in C and its parse tree.*

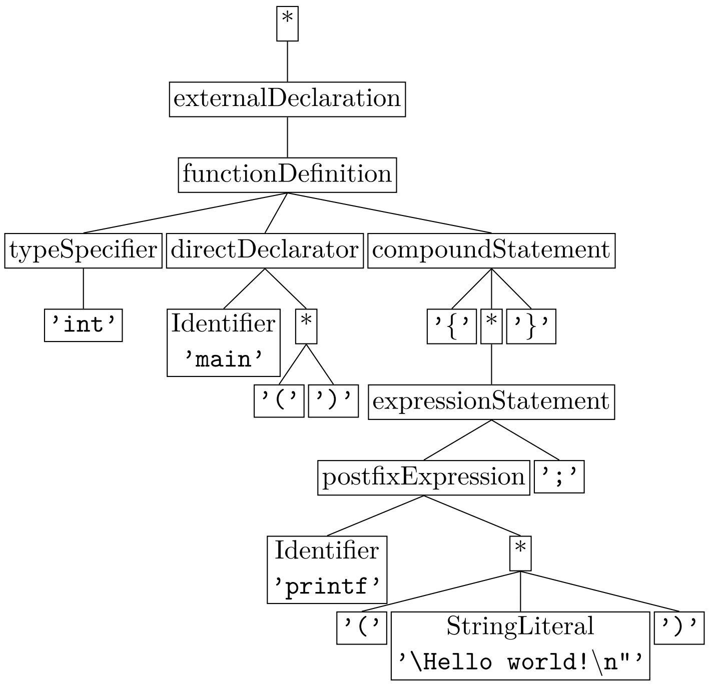
generated by a parser using a context-free grammar of the C programming language and preprocessed for compactness (most notably, squeezing and recursion flattening have been applied). Unfortunately, none of the HDD-based algorithms presented in Section 2 can reduce this test case any further (especially 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$^{\mathrm{r}}$, 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, another approach is proposed, called *hoisting*.

Recurring structures are present 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. In the tree of Figure 7.1(b), there is one pair of such subtrees, those rooted at nodes labeled as *compoundStatement*. Figure 7.2(a) 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 7.2(b)), it becomes apparent that, in this case, this transformation was indeed useful and we got a smaller and still interesting test case. The testing function has to confirm (or reject) whether such a transformation keeps the resulting test case interesting.

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 7.1 does not seem to contradict such arguments. However, Figure 7.3 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. This program shall be taken 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, 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 Figure 7.4(a) contains only the HDD-reduced Java program, which also shows precisely what HDD can and 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



**(a)** *Parse tree minimized with hoisting applied to keep printing the "Hello world!" message.*

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

**(b)** *The C program serialized from the minimized tree.*

**Figure 7.2:** *Motivational Example for Hoisting.*

```java
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);
  }
}
```

**Figure 7.3:** *Java program that prints the rounded value of $\pi$ in a locale-specific format or throws an exception.*

structures that are in ancestor-descendant relation in the tree. The result of the combined application of the two techniques is presented in Figure 7.4(b). *formatParts* could be constructed arbitrarily complex, making the theoretical potential of hoisting considerably higher than the removal of anecdotical semicolons or curly braces. Hoisting cannot achieve this alone, as it “only” moves subtrees higher up the tree, but it has to cooperate with HDD.

## 7.1 Transformation-based Minimization

To formalize the ideas motivated and described above, the notations and terminology of minimizing Delta Debugging (as given in Section 2) were extended to introduce transformation-based minimization. In the context of DDMIN, a test case is always composed of a subset that contains elements of the initial configuration. The testing function is also defined for the subsets of $c_{✗}$ only. However, the outcome of a program composed of a set of elements must be determined, even if some of them are not part of the initial configuration. 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, the definitions of DDMIN [40] are generalized as follows.

Let $D$ denote the set of all potential test case elements, and let $\delta \in D$ denote

```java
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]));
  }
}
```

**(a)** *Program minimized with HDD to keep the program throwing an uncaught exception if an unsupported locale is specified on the command line.*

```java
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]);
  }
}
```

**(b)** *Program minimized with with hoisting applied before HDD*

**Figure 7.4:** *Minimization of the Java program that print the rounded value of $\pi$ (see Figure 7.3).*

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. The following definitions and notations are used for transformations:

A function $t : D \to D$ is a transformation of test case elements, and the identity transformation is $id_D : D \to D; \delta \mapsto \delta$. The application of a transformation to configurations is defined 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 presented examples, 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, 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, a $\|\cdot\|$ measure is expected 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, the replacements applied to the elements of the initial configuration must be maximized (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, the 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 7.5 wraps up this subsection and formalizes the transformation-based minimizing algorithm TMIN$^\tau$, worded in the likeness of DDMIN.

## 7.2 Variants of Hoisting and HDD

The transformation-based minimizing algorithm gives a framework to formulate hoisting as a transformation of tree nodes. More precisely, those nodes in the tree representation of the input that can act as replacement candidates for their ancestors must be defined. The formula in Figure 7.6, $\chi(n)$, 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

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 $c_{✗} = \{\delta_1, \dots, \delta_n\} \subseteq D$, 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 7.5:** *The Transformation-based Minimizing Algorithm.*

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 2.2, 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.

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

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

**Figure 7.6:** $\chi(n)$*, the potentially hoistable descendants of node* $n$*.*

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.

Now, with the help of TMIN$^{\chi}$, a hierarchical algorithm can be introduced, 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, with further nodes getting higher priority. The pseudocode of the algorithm is presented in Figure 7.7(a). The structure of Hoist is similar to 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 in the example of Figures 7.4 and 7.3, 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, they can be combined as well. *E.g.,* the bodies of the loops can be interlaced, performing both the DDMIN and TMIN$^{\chi}$-based minimization at each iteration. One way to formulate this idea is shown in Figure 7.8, where HDD and Hoist are interlaced in the algorithm named HDDH.

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 be defined. These are given in Figures 7.7(b), 7.7(c), and 7.7(d), and are named Hoist$^{\mathrm{r}}$, Coarse Hoist, and Coarse Hoist$^{\mathrm{r}}$, respectively. Similarly, we can create new combined algorithms from HDD$^{\mathrm{r}}$, and Hoist$^{\mathrm{r}}$ (HDDH$^{\mathrm{r}}$), from Coarse HDD, and Coarse Hoist (Coarse HDDH) and from HDD$^{\mathrm{r}}$ and Coarse Hoist (Coarse HDDH$^{\mathrm{r}}$). These combinations are trivial following the example of HDDH, therefore, they are not shown to avoid unnecessary repetition.

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

**(a)** *Hoisting.*

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

**(b)** *Recursive Hoisting.*

```
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
            $hoisting \leftarrow \text{TMIN}^{\mathcal{X}}(nodes)$
            $transform(input\_tree, level, hoisting)$
        end if
        $level \leftarrow level + 1$
        $nodes \leftarrow tagNodes(input\_tree, level)$
    end while
end procedure
```

**(c)** *Coarse Hoisting.*

```
procedure CoarseHoist$^{\text{r}}$($root\_node$)
    $queue \leftarrow \langle root\_node \rangle$
    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
            $hoisting \leftarrow \text{TMIN}^{\mathcal{X}}(nodes)$
            $transformChildren(current\_node, hoisting)$
        end if
        $append(queue, tagChildren(current\_node))$
    end while
end procedure
```

**(d)** *Recursive Coarse Hoisting.*

**Figure 7.7:** *Proposed Hoisting algorithm and its variants.*

## 7.3 Evaluation

The experimental setup described in Chapter 3 was used to evaluate the effects of the proposed algorithms. The workstation used to conduct the experiments was 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 5.11.0, and running the experiments only.

Four experiments were conducted, one for each variant of HDD and hoisting (original, recursive, coarse, and coarse recursive). Four different combinations of HDD and hoisting were used as summarized in Table 7.1. The columns are organized as follows:

- **Algorithm:** the fixed-point iteration of the variant without any hoisting

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

**Figure 7.8:** *The Hierarchical Delta Debugging and Hoisting algorithm.*

(*e.g.*, HDD*) acted as the baseline,

- **Preprocessing:** hoisting was applied as a preprocessing step to hierarchical delta debugging (*e.g.*, Hoist*+HDD*),
- **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*).

**Table 7.1:** *Hoisting and the variants of HDD*

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

**HDD.** In the first experiment, HDD*, Hoist* + HDD*, HDDH*, and Hoist* + HDDH* algorithms were compared (see Figures 2.2(a), 7.7(a), and 7.8). 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 at the time of publication of the related article.) The average effect on size was 37.50%, 39.04%, and 40.16%, 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 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, hoisting could have a positive effect on the number of overall test case evaluations, but not necessarily. HDDH* performed the minimization of 18 inputs faster than the baseline HDD*, but in those approaches where hoisting was a preprocessing step, this improvement was only visible in 14 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 Tables 7.3 and 7.4.

**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 2.2(b) and 7.7(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 Tables 7.5 and 7.6.

**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 2.3(a) and 7.7(c)). When hoisting acted as a preprocessing step, algorithm combinations produced smaller outputs 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, 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 Tables 7.7 and 7.8.

**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 2.3(b) and 7.7(d)). Tables 7.9 and 7.10 contain 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 7.9 visualizes the raw data from Tables 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, and 7.10. 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

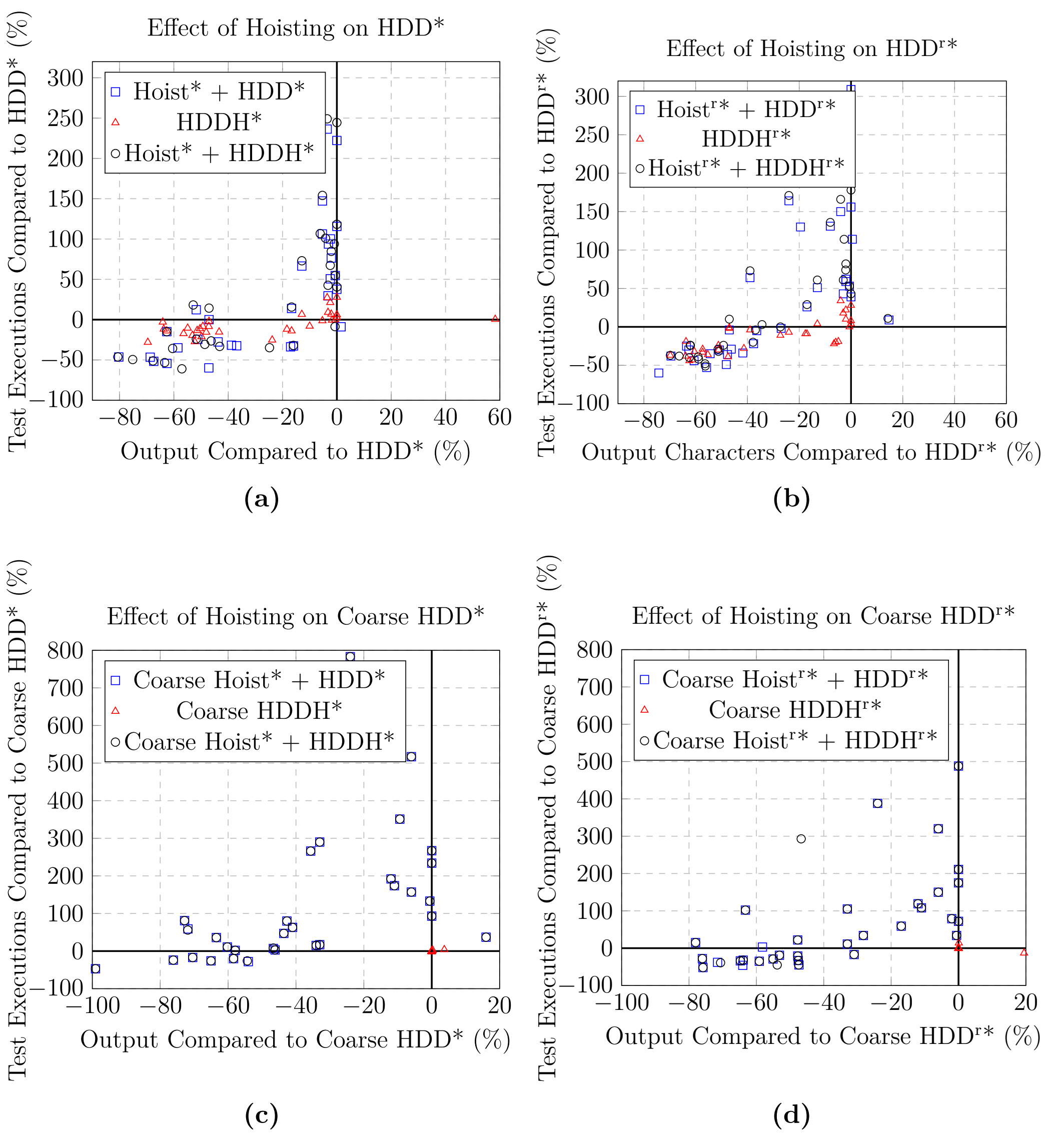
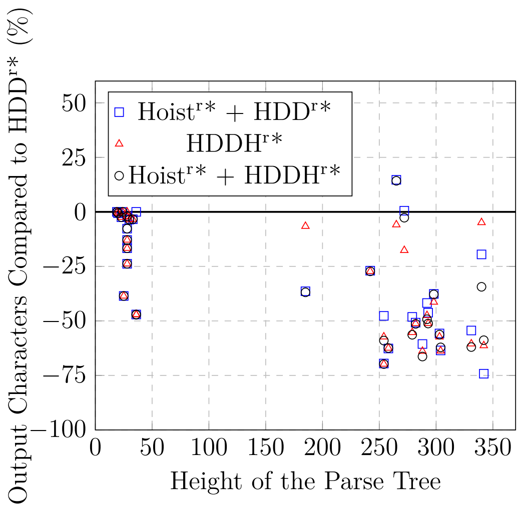


**Figure 7.9:** *The effect of hoisting on variants of HDD.*

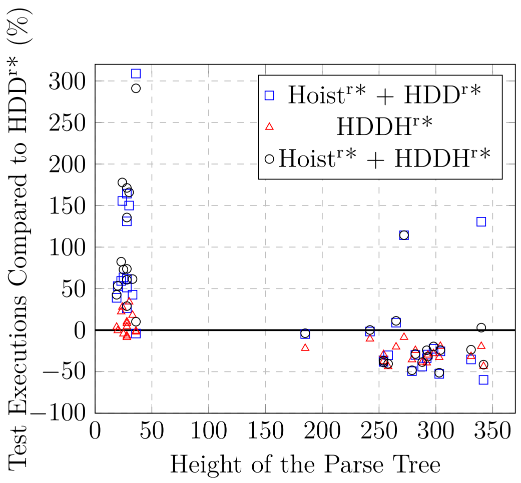
reduction. Changes in the size of the output are represented along the horizontal axis, while changes in the test executions are represented along the vertical axis (all relative to the size and test step count of the baseline). Marks in the "bottom left" quadrant (a.k.a. quadrant III) are considered 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 7.9(a) and 7.9(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 as 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 7.9(c) and 7.9(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 7.2 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 7.1, with variant names replaced with values that represent the effect of hoisting compared to the respective baseline, which is in the first column. Table 7.2(a) 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 7.2(b) 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 HDD variants. When using hoisting both as a preprocessing step and interlaced with the HDD variants, the number of testing steps increased by 72.97%. However, interestingly, using only interlaced hoisting required 4.48% *fewer* tests.



**(a)**



**(b)**

**Figure 7.10:** *Effect of Hoisting on HDD$^{r}$* as a Function of* Height *of the Parse Tree.*

In the above experiments, the number of test executions has been used as the 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 the experiments. Moreover, roughly the same rate of deterioration and improvement could be observed in execution times as in test steps. It must also be noted that if it is accepted 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 the experiments. When the baseline execution time was a few 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.

Tables 3.1 and 3.2 shows the properties of the inputs used for benchmarking, thus providing opportunities to investigate whether they affect hoisting.

**Table 7.2:** *Averaged impact of hoisting on different HDD variants*

**(a)** *Differences in Output Sizes (%)*

| **Algorithm** | **Preprocessing** | **Interlacing** | **Preprocessing & Interlacing** |
|---|---:|---:|---:|
| HDD* | -27.49% | -28.31% | -29.07% |
| 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* | +43.26% | -5.60% | +47.21% |
| 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% |

It turned out that the height of the parse tree had an effect on the efficiency of hoisting, as shown in Figure 7.10. The chart is similar to Figure 7.9, *i.e.*, the vertical axis shows the relative difference compared to the output (Figure 7.10(a)) or the testing steps (Figure 7.10(b)) of the baseline HDD$^{r}$* algorithm. But now the horizontal axis represents the height of the parse tree built from the test cases after squeezing and flattening [19, 20]. 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 7.10(a) confirms the first observation, Figure 7.10(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.)

So far, the discussion was only about relative changes compared to the

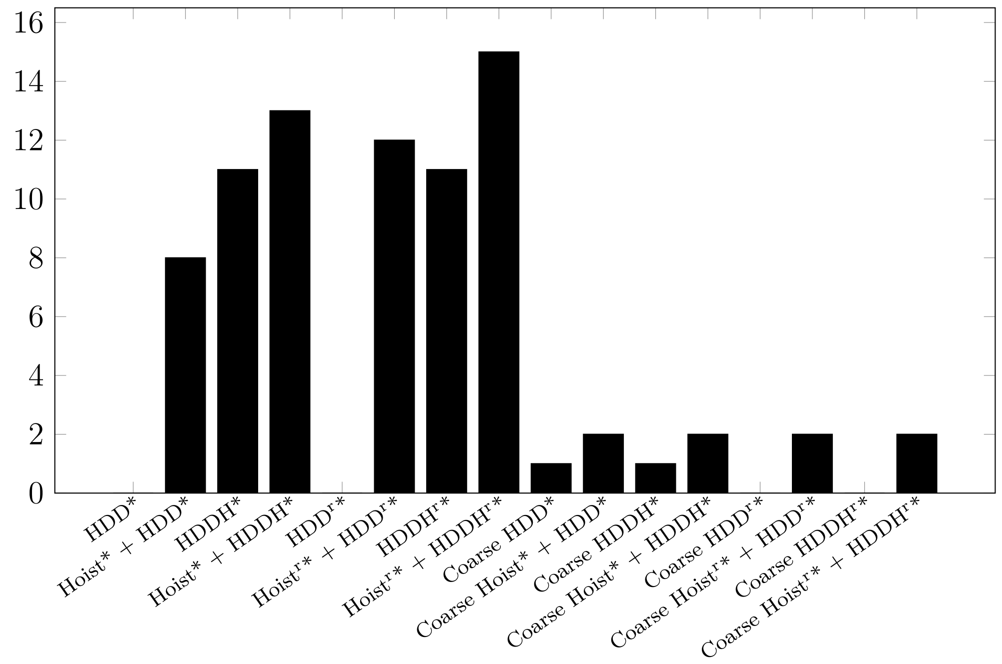


**Figure 7.11:** *Best-performing Hoisting Algorithms Based on Output Characters.*

baseline algorithms. However, the reader might be interested in using the best-performing variant in an absolute sense and this is what Figures 7.11 and 7.12 are 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 7.11 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 7.2(a)), however, in an absolute manner the Coarse variants fall behind the others effectiveness-wise. 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

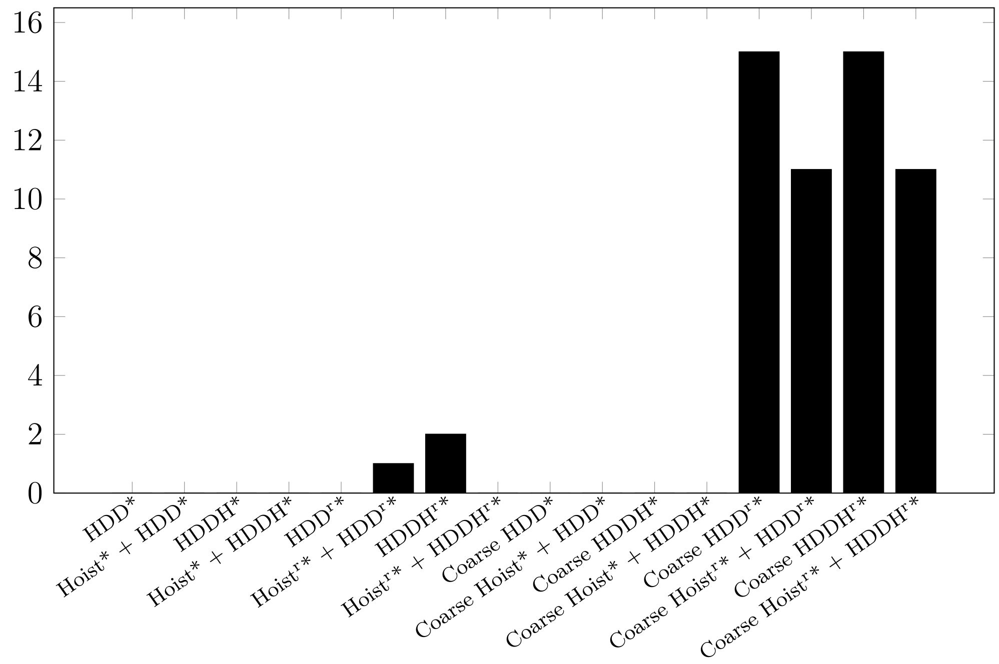


**Figure 7.12:** *Best-performing Hoisting Algorithms Based on Number of Test Executions.*

results, if a test must be reduced to the smallest possible size, the suggested mode is 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 7.12 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.

## 7.4 Conclusions

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. Therefore, the Transformation-based Minimization describes an algorithmic framework that enables transformations that cannot only remove but also replace elements in the initial configuration. We have already defined such a transformation (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.

Based on the experimental data and observations above, we can conclude the contributions of this chapter:

1. 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.

2. The effects of hoisting to HDD and HDD$^r$ are similar: the majority of the test cases could be reduced further with hoisting.

3. Coarse HDD and Coarse HDD$^r$ show similar patterns to the non-coarse variants with respect to 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.

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.

## 7.5 Raw 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 7.3:** *Hoisting and HDD Output Sizes*

| **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%)* |

**Table 7.4:** *Hoisting and HDD 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 7.5:** *Hoisting and HDD$^r$ Output Sizes*

| **Test** | **HDD$^{r*}$** | **Hoist$^{r*}$+HDD$^{r*}$** | | **HDDH$^{r*}$** | | **Hoist$^{r*}$+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%)* |

**Table 7.6:** *Hoisting and HDD$^r$ Number of Test Executions*

| **Test** | **HDD$^{r*}$** | **Hoist$^{r*}$+HDD$^{r*}$** | | **HDDH$^{r*}$** | | **Hoist$^{r*}$+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 7.7:** *Coarse Hoisting and HDD Output Sizes*

| **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%)* |

**Table 7.8:** *Coarse Hoisting and HDD 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 7.9:** *Coarse Hoisting and HDD$^{r}$ Output Sizes*

| **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%)* |

**Table 7.10:** *Coarse Hoisting and HDD$^r$ 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%)* |

*"Working hard for something we don't care about is called stress; working hard for something we love is called passion."*

— Simon Sinek

# A
# Summary

Software maintenance is a diverse field, and this study only focused on a small part of it: what happens after a new bug is found but before it is reported. When a new bug is found, the sequence of events that can reproduce it might be noisy, containing both relevant and irrelevant information related to the reproduction. It is better for everyone to start debugging with a clean test case that reproduces the bug, however, manually selecting the events that are absolutely necessary for its reproduction is also a time-consuming task. Fortunately, test cases can be minimized automatically, and this study focused on automatic test case reduction algorithms and their optimization possibilities.

Two discipline defining algorithms are evaluated in detail, presented their weaknesses and proposed optimizations to them. First, the minimizing Delta Debugging algorithm was optimized from different perspectives. When the proposed optimizations had a potential effect on Hierarchical Delta Debugging (as HDD uses DDMIN as its utility to remove subtrees), we evaluated the proposals on HDD as well. Then, investigated the potential behind different tree-transformations (other than pruning) with HDD.

The author identifies four main findings, which are listed below:

1. **Cache Optimizations**: Defined, implemented, and evaluated three cache optimizations for automatic test case reduction algorithms,
2. **Iterating the Minimizing Delta Debugging Algorithm**: Defined, implemented, and evaluated the fixed-point iteration of DDMIN that achieves smaller outputs,
3. **Parallel Optimizations of DDMIN***: Defined, implemented, and evaluated a new parallel variant of the DDMIN algorithm that can perform reduction faster than before,
4. **Extending Hierarchical Delta Debugging with Hoisting**: Defined, implemented, and evaluated the transformation-based minimization framework for hierarchical reduction, and an example transformation, the Hoisting.

Some chapters focused on making the output of the algorithms smaller while others focused on making the reduction process more lightweight.

## 1. Cache Optimizations

The main purpose of using cache memory in test case reduction algorithms is to avoid running the same test multiple times as the algorithm tries different configuration combinations. On the other hand, the reduction should somehow be completed, even if we run out of resources (RAM). Several techniques are available for cache replacement, the most widespread algorithms for it are Least Frequently Used (LFU), Most Frequently Used (MFU), and Least Recently Used (LRU), however, these classic techniques do not make use of the knowledge of the underlying algorithms and evict elements from the cache that might be needed later. Therefore, the caching solutions of minimizing Delta Debugging (DDMIN) and Hierarchical Delta Debugging (HDD) were investigated, and based on the preliminary research, the "content-based" technique was chosen to work on as it performed the best with both algorithms.

The contributions of this part:

1. Found that the cache utilization and scaling are suboptimal: DDMIN determined the outcome of its configurations via cache memory only in 3% of the cases, while HDD utilized the cache better, the actual testing of 21% of the

configurations could be avoided. It did not scale well for either algorithm: DDMIN consumed up to 53 MB of memory for reducing a 4 kB sized input, while HDD required 4 GB of RAM to reduce a 0.44 MB sized test in the worst case.

2. Three optimizations were proposed to reduce the memory footprint of caches used in test case minimization:
   (a) add only *passing* (✓) tests to the cache,
   (b) when a new *failing* (✗) test case is found, evict cache entries of bigger test cases, and
   (c) instead of storing the serialized test contents, store their *hashed* value (fixed-width keys instead of variable-width).
3. With the optimizations combined, DDMIN requires 96% and HDD requires 85% less memory compared to the baseline implementation. Supporting the scalability issue, the size of the input had an effect on the results: on JRTS (smaller tests), the average improvement was 63.19%, while on PTS (larger inputs), it was 99.93%. Furthermore, as a side effect, the reduction becomes faster by 9.9% with DDMIN. In our experiments, the result of the reductions did not change after the optimizations.

The results, which are based on the [37] publication are presented in Chapter 4.

## 2. Iterating the Minimizing Delta Debugging Algorithm

If test case reducers have to work without information about the input structure, suboptimal results may be produced. It is easy to show what DDMIN can remove from its input and what structures cannot be handled by the algorithm. We analyzed inputs (source code for various compilers and execution engines) that cannot be properly reduced by DDMIN and proposed a potential solution to the problem. We formalized the fixed-point iteration of DDMIN (denoted as DDMIN*) and then investigated whether it improves the effectiveness of the algorithm. We evaluated the fixed-point iteration in two slightly different settings.

First, the reduction of test cases was performed with character level granularity on a smaller test suite; the output became smaller by 68% on average. Then, the reduction was performed with line granularity, and the experiments show that DDMIN* can produce 48% smaller outputs on average (69% on a larger test suite and 19% on a smaller one). The price of this improvement is an increase in the number of steps, which was 66% on average. A "combined", two-pass reduction was then performed where test cases were first reduced with line granularity, and then these intermediate results were reduced further with character granularity for fine-tuning. DDMIN* outperformed DDMIN even with this setting, and was able to reduce inputs by an average of 46%. Surprisingly, some inputs could be reduced faster with DDMIN*, as the line-level reduction produces results in a reasonable amount of steps, then the character-level reduction can work further from this smaller input configuration.

Encouraged by the promising results, we compared the output of DDMIN* with the output of HDD* to see whether a structure-unaware algorithm could compete with a "more clever" one. In terms of required testing steps, the answer is simply no; however, in terms of size, DDMIN* has brought the results much closer, from a 9 times larger output (DDMIN) to an only 3 times larger one.

The contributions of this part:

1. Formalization of the fixed-point iteration of DDMIN, denoted as DDMIN*,

2. DDMIN* is most effective with character-level granularity, however, character-level reduction can be unacceptably slow for "large" inputs and line-level reduction leaves unnecessary parts in its output. Therefore, we used a combined approach, where DDMIN* produced 46% smaller outputs. Furthermore, the two-pass reduction produced 54% smaller outputs than the line-level approach in our experiments, on average.

3. DDMIN* incurs an additional cost (number of testing steps), which appears in most cases. This additional cost is related to the size of the test case, but it does not grow beyond all limits. The effectiveness of the reduction shows a similar pattern: the larger the input configuration, the larger the potential to reduce. If the input configuration has some superfluous items, DDMIN* can reduce it further regardless of its size.

4. We have compared the output of DDMIN* with the output of HDD*, to see if a structure-unaware algorithm can compete with a "more clever" one. In terms of required testing steps, the answer is simply no; however, DDMIN* brought the results much closer to each other, from a 9 times larger output (DDMIN) to an only 3 times larger.

The results, which are based on the publications [32, 35] are presented in Chapter 5.

## 3. Parallel Optimizations of DDMIN*

The previous two parts tried to make the reduction more lightweight and make its output smaller. In this part, we have investigated whether it was possible to make DDMIN itself work faster without compromising its minimality guarantees. A technique that has already been proven useful for speeding up DDMIN is parallelization, and we have investigated whether it was possible to make *parallel DDMIN* even faster without losing the 1-minimal property of its output.

First, the stability issues of parallel DDMIN are discussed: if multiple *failing* configurations are found in a parallel testing window (how many tests are checked concurrently), then the algorithm becomes unstable: it will choose among the interesting configurations based on which produced its *fail* outcome first. Different test reductions can yield different outcomes, which is not appropriate for carrying out reproducible experiments. The "reduce to subset" and "reduce to complement" phases iterate through configurations in a forward or backward syntactic order; it is known which configuration *should* be investigated first. The following changes are made in order to stabilize the algorithm: if a *fail* is found in a parallel testing window, then the active *test* executions should be awaited (*i.e.*, computation results are discarded). If multiple *fail*s are found, the syntactic order is considered when choosing which one to reduce further.

When multiple *fail*s are found and the algorithm chooses one of them based on the iterator, the results from other configurations are discarded, even if they could have been useful. This results in superfluous test executions on configurations that have already been tested (and *fail*ed). The following strategy helps to minimize the number of test executions: If a testing window has multiple *fail*s, then it is worth trying to combine those configurations that yielded them and check whether this combination also results in a *fail*. If yes, then multiple test executions are saved in one step. If not, select the first *fail* (based on the syntactic order) and try to combine the other interesting configurations one by one. This case can also save testing steps as only configurations with a *fail* outcome are retested instead of the whole testing window in the next parallel loop iteration.

We have presented a modified parallel DDMIN – called *GreeDDy* –, evaluated on a subset of a publicly available dataset and found that *GreeDDy** could save 31% of the testing steps of DDMIN* which resulted in 40% less runtime.

The contributions of this part:

1. Investigated the stability issues of DDMIN* and provided a stabilization approach for it.

2. Made the parallel execution of DDMIN* even faster, exploiting the potentials of the already-seen testing windows (named *GreeDDy**).

3. Presented our idea where multiple test executions with *fail* outcomes could be merged to save further retesting, and found that *GreeDDy** could save 30.68% of the testing steps of DDMIN* which resulted in 39.88% less runtime. The output of *GreeDDy** usually got smaller, however, the effects of the algorithm on the output are negligible.

The results, which are based on the [36] publication are presented in Chapter 6.

## 4. Extending Hierarchical Delta Debugging with Hoisting

Although Hierarchical Delta Debugging 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 to reduce the number of nodes that have to be considered, by collapsing multiple nodes into one for the same reason, or by rotating recursive structures of the tree to reduce its height. However, these transformations do not change the core structure of the tree, the test case serialized from the preprocessed tree will still be the same as the original input.

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

The contributions of this part:

1. Formalization of the Transformation-based Minimization framework and an example transformation: hoisting. The intention behind hoisting is that a

subtree may be replaced by another without losing syntactic correctness if the roots of the subtrees are in an ancestor-descendant relationship.

2. Evaluation of hoisting in several configurations. First, hoisting was applied as a preprocessing step to hierarchical delta debugging (*e.g.*, Hoist*+HDD*), then, hoisting was interlaced with hierarchical delta debugging (*e.g.*, HDDH*), and as stand-alone hoisting and the interlaced algorithm are not mutually exclusive, they can be used in sequence (*e.g.*, Hoist*+HDDH*).

3. On real-world inputs, hoisting combined with HDD gives generally smaller, or at least as small outputs as HDD alone. Bigger outputs are rare. Minimized test cases can be as small as $1/5$ of the output given by traditional HDD.

4. The effects of hoisting to HDD and HDD$^r$ are similar: the majority of the test cases could be reduced further with hoisting.

5. Coarse HDD and Coarse HDD$^r$ show similar patterns to the non-coarse variants with respect to 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.

6. 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.

The Hoist* + HDDH* and Hoist$^r$* + HDDH$^r$* algorithm variants produced the smallest output among the ones tested, and the Coarse variants performed the reduction requiring the fewest steps (at the cost of bigger outputs). The results, which are based on the publications [33, 34] are presented in Chapter 7.

***The Author's Contributions***

The author had a decisive role in the design, implementation and evaluation of a significant proportion of the above presented findings.

1. **Cache Optimizations**: The author analyzed the state-of-the-part caching solutions that have been used in reduction, then designed and implemented three optimizations for reducing the memory footprint of the reduction.

**Table A.1:** *Summary of thesis topics and corresponding publications*

| | [37] | [32] | [35] | [36] | [33] | [34] |
|---|---|---|---|---|---|---|
| 1 | ● | | | | | |
| 2 | | ● | ● | | | |
| 3 | | | | ● | | |
| 4 | | | | | ● | ● |

Then, he evaluated the effects of proposals with different reduction approaches, on multiple test suites.

2. **Iterating the Minimizing Delta Debugging Algorithm**: The author designed and prototyped the fixed-point iteration of the DDMIN algorithm.

3. **Parallel Optimizations of DDMIN***: The author analyzed the weaknesses of the parallel DDMIN, then designed, implemented and evaluated a solution to it.

4. **Extending Hierarchical Delta Debugging with Hoisting**: The author investigated the structure of the inputs (abstract syntax trees) of HDD searching for optimization possibilities. He found that identically labeled nodes can be replaced without loosing the syntactic correctness. Then, he prototyped the transformation-based minimization framework, implemented the hoisting as an example transformation, and evaluated it on publicly available test suites.

Furthermore, the supporting replication package has been published at the time of each publication. The author has been responsible for the redesign and implementation of the algorithms that stand their ground in the world of open-source. The publications related to the thesis points are the following:

[37] **Dániel Vince** and Ákos Kiss. Cache Optimizations for Test Case Reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442-453, Guangzhou, China (Virtual), December 2022. IEEE.

[32] **Dániel Vince**. Iterating the Minimizing Delta Debugging Algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57-60, Singapore, November 2022. ACM.

[35] **Dániel Vince** and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. In *Journal of Software: Evolution and Process*, 2024. Wiley.

[36] **Dániel Vince** and Ákos Kiss. GreeDDy: Accelerate Parallel DDMIN. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '24)*, pages 1-4, Vienna, Austria, September 2024. ACM.

[33] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending Hierarchical Delta Debugging with Hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60-69, Madrid, Spain (Virtual), May 2021. IEEE.

[34] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of Hierarchical Delta Debugging. In *Journal of Software: Evolution and Process*, 34(11):e2483:1-e2483:26, November 2022. Wiley.

The author notes that although the results presented in this thesis are his major contribution, the term *we* is used instead of *I* for self-reference to acknowledge the contributions of the co-authors of the papers that this thesis is based on.

# B
# Összefoglalás

A szoftverkarbantartás egy szerteágazó tudományág és jelen tanulmány annak egy kis részével foglalkozik: mi történik azután, hogy egy hibát megtaláltak, de még azelőtt, hogy bejelentenék. Amikor egy új hibát találnak, a reprodukcióhoz szükséges eseménysorozat a releváns részek mellett zajt (irreleváns részeket) is tartalmazhat. Mindenkinek jobb, ha egy letisztult tesztesettel kezdi meg a hibakeresést, viszont manuálisan kiválogatni a releváns részeket egy tesztesetből is időigényes feladat. Szerencsére a tesztesetek minimalizálása automatizálható és a tanulmány az automatikus teszteset-redukciós algoritmusokra és azok optimalizációs lehetőségeire fókuszált.

Két, a diszciplínát meghatározó algoritmus részletesen ki lett értékelve, bemutatásra kerültek a gyengeségeik és optimalizációk a kiküszöbölésükre. Először a minimizing Delta Debugging lett optimalizálva különböző perspektívából. Amikor az optimalizációnak potenciális hatása lehetett a Hierarchical Delta Debugging algoritmusra (mivel a HDD a DDMIN-t használja a részfái törléséhez), ott kiértékeltük az optimalizációt a HDD-n is. Majd megvizsgáltuk a lehetőséget különböző fa-transzformációk alkalmazására (a törlésen kívül) a HDD segítségével.

A szerző négy fő megállapítást azonosít, melyek a következők:

1. **Cache Optimizations**: Három gyorsítótár-optimalizálást definiált, valósított meg és értékelt ki az automatikus teszteset-csökkentési algoritmusokhoz,
2. **Iterating the Minimizing Delta Debugging Algorithm**: Definiálta a DDMIN fixpontos iterációját, amely kisebb kimenet eredményez,
3. **Parallel Optimizations of DDMIN***: Definiálta a DDMIN algoritmus egy új párhuzamos változatát, amely gyorsabban képes végrehajtani a redukciót,
4. **Extending Hierarchical Delta Debugging with Hoisting**: Definiálta a transzformáció-alapú minimalizálási keretrendszert és egy példa transzformációt, a Hoistinget.

Néhány fejezet azzal foglalkozott, hogy a redukció kimenete kisebb legyen, még mások a redukciós folyamat könnyedebbé tételét boncolgatta.

## 1. Cache Optimizations

A gyorsítótár használatának célja a teszteset redukciós algoritmusokban az, hogy elkerüljük ugyanazon tesztesetek újrafuttatását miközben az algoritmus különböző konfigurációs kombinációkat próbál ki. Másrészről a redukciónak akkor is végbe kellene mennie, amikor az erőforrásokból kifogyunk (RAM memória). Több megoldás is létezik a gyorsítótár menedzselésére, a legelterjedtebb algoritmusok a Least Frequently Used (LFU), a Most Frequently Used (MFU) és a Least Recently Used (LRU), viszont ezek a klasszikus megoldások nem veszik figyelembe az algoritmusok sajátosságait és törölhetnek olyan elemeket a gyor-sítótárból, amikre később szükség lehet. Ezért megvizsgáltuk a minimizing Delta Debugging (DDMIN) és a Hierarchical Delta Debugging (HDD) algoritmusok gyorsítótár megoldásait. Az előzetes vizsgálatok alapján a tartalom alapú megoldás lett kiválasztva további javításra, mivel ez bizonyult a leghatékonyabbnak mindkét algoritmus számára.

A fejezet kontribúciói:

1. Kiderült, hogy a gyorsítótár felhasználása és a skálázódása sem optimális: DDMIN a kipróbált konfigurációinak csupán 3%-át határozta meg gyorsítótárból, míg a HDD jobban teljesített, a kipróbált konfigurációinak 21%-át tudta gyorsítótárból meghatározni. Ami még ennél is rosszabb, egyik algoritmus skálázódása sem volt megfelelő: DDMIN 53MB memóriát használt

fel egy 4kB méretű teszteset minimalizálásához, míg HDD-nek 4GB RAM volt szükséges egy 0.44MB méretű tesztesethez.

2. A tartalom alapú gyorsítótár memóriaigényének csökkentése érdekében a következő három optimalizációt javasoltuk:
   (a) csak a hibát nem reprodukáló teszteseteket adjuk hozzá a gyorsítótárhoz,
   (b) amikor egy reprodukáló tesztesetet talál az algoritmus, minden annál nagyobbat töröljünk ki a gyorsítótárból, és
   (c) ahelyett, hogy a teszteset tartalmát tárolnánk, tároljuk annak egy transzformált (hashelt) változatát (azonos hosszúságú kulcsok a változó hosszúságúak helyett).
3. A javasolt optimalizációkkal a DDMIN 96%-kal és a HDD 85%-kal kevesebb memóriát használt az összehasonlításképpen használt implementációhoz képest. A skálázódási problé-mát alátámasztva a bemenet mérete befolyásolta az eredményeket: az átlagos memóriajavulás 63% volt kisebb teszteseteken, míg 99% volt nagyobbakon. Mellékhatásként a redukció 10%-kal gyorsabb lett a DDMIN-t használva. A kísérleteinkben a redukció kimenete nem változott az optimalizációk hatására.

Az eredmények a 4. Fejezetben vannak bemutatva, melyek a [37] tanulmányra épülnek.

## 2. Iterating the Minimizing Delta Debugging Algorithm

Amikor a tesztesetredukáló eszközöknek anélkül kell elvégezniük a feladatot, hogy rendelkeznének információval a bemenetük struktúrájáról, szuboptimális eredmények keletkezhetnek. Könnyű olyan példát konstruálni, amely szemlélteti, hogy a DDMIN mit tud kitörölni és milyen struktúrákat nem tud kezelni. Megvizsgáltuk azokat az eseteket, amiket a DDMIN nem tud megfelelően redukálni (forráskód formátumú bemenetek különböző fordítóprogramokhoz és végrehajtómotorokhoz) és javasoltunk egy lehetséges megoldást a problémára. Formalizáltuk a fixpont iterációját a DDMIN-nek (DDMIN*-gal jelöltük), majd megvizsgáltuk vajon javítja-e az algoritmus hatékonyságát. Ezután kiértékeltük a fixpont iterációt két, némileg különböző esetben.

Először a tesztesetek karakteralapú bontásban kerültek redukálásra egy kisebb teszthalmazon, ahol a kimenet 68%-kal csökkent átlagosan. Majd soralapú redukció lett vizsgálva, a kísérletek alapján a DDMIN* 48%-kal kisebb kimenetet tud előállítani (69%-kal kisebbet egy nagyobb méretű teszthalmazon és 19%-kal kisebbet egy kisebben). A kimenet méretének csökkentése megnövelte a szükséges lépésszámot, mely 66%-kal növekedett átlagosan. Végül egy “kombinált”, kétfázisú redukciót vizsgáltunk, ahol a teszteseteket először soralapon redukáltuk, majd a köztes eredmények karakteralapú redukcióval lettek tovább csökkentve, mint egy finomhangolási lépés. A DDMIN* ebben a kísérletben is felülmúlta az összehasonlításképpen használt DDMIN-t és tovább tudta redukálni a bemeneteit 46%-kal. Meglepő módon néhány teszteset redukciója gyorsabban lefutott, ahogyan a soralapú redukció ésszerű idő alatt produkált eredményeket, majd a karakteralapú redukció ezen a kisebb konfiguráción tudott dolgozni.

Az ígéretes eredményeken felbuzdulva összehasonlítottuk a DDMIN* és a HDD* eredmé-nyeit, hogy lássuk vajon egy struktúrát figyelmen kívül hagyó algoritmus tud-e versenyezni egy nála okosabbal. A szükséges tesztlépések számát tekintve a válasz egyszerűen nem, viszont a DDMIN* sokkal közelebb hozta egymáshoz a kimeneti méreteket: kilencszer nagyobb eredmények helyett (DDMIN) csak háromszor nagyobbat csinált. Még mindig van mit javítani azokban a helyzetekben, amikor a bemeneti struktúra hiányzik vagy gyorsan változik.
A fejezet kontribúciói:

1. A DDMIN fixpont iterációjának formalizálása, melyet DDMIN*-gal jelöltünk.

2. A DDMIN* karakteralapú redukcióval a leghatékonyabb, viszont ez gyakorlati szempontból elfogadhatatlanul lassú lehet nagy tesztesetek esetében. A soralapú redukció pedig irreleváns részeket hagy a kimenetében. Ennélfogva egy kombinált megközelítést alkalmaztunk, ahol a DDMIN* 46%-kal kisebb kimenetet tudott előállítani. Továbbá a kétfázisú redukció átlagosan 54%-kal kisebb kimenetet eredményezett a soralapú redukcióhoz viszonyítva a kísérleteink során.

3. DDMIN* használata többletköltséggel jár, a tesztvégrehajtások száma növekszik, ami a legtöbb esetben fellelhető. Ez a többletköltség a bemenet nagyságától is függ, viszont nem nő minden határon túl. A redukció eredményeként előálló kimenet is hasonló tendenciát mutat: minél nagyobb a bemeneti konfiguráció, annál nagyobb a lehetőség a redukcióra. Amennyiben a bemenet tartalmaz felesleges elemeket, a DDMIN* tovább tudja redukálni azt a mérettől függetlenül.

4. Összehasonlítottuk a DDMIN* és a HDD* kimeneteit, hogy megvizsgáljuk vajon egy struktúrát nem ismerő algoritmus versenyezhet-e egy nála okosabbal. A redukcióhoz szükséges tesztvégrehajtások számában nem, viszont a DDMIN* sokkal közelebb hozta egymáshoz a kimeneti eredményeket: kilencszer nagyobb eredmények helyett (DDMIN) csak háromszor nagyobbat csinált.

Az eredmények a 5. Fejezetben vannak bemutatva, melyek a [32, 35] tanulmányokra épülnek.

## 3. Parallel Optimizations of DDMIN*

Az előző két fejezet megpróbálta a redukció folyamatát könnyedebbé és a kimenetét kisebbé tenni. Ebben a részben megvizsgáltuk vajon lehetséges-e magát a DDMIN-t gyorsabbá tenni anélkül, hogy az algoritmus minimalitásra vonatkozó garanciáit elvesztenénk. Egy, már a gyakorlatban is bizonyított technika a párhuzamosítás, ezért azt vizsgáltuk, hogy gyorsabbá tudjuk-e tenni a párhuzamos redukciót az 1-minimalitás megtartása mellett.

Először a párhuzamos DDMIN stabilitási problémáit ismertettük: amennyiben több reprodukáló konfigurációt találunk egy tesztelési ablakban (azaz hány konfiguráció van tesztelve párhuzamosan), akkor az algoritmus viselkedése instabil lesz: azt a reprodukáló konfigurációt választja, amely időben először adott vissza eredményt. Emiatt különböző redukciós folyamatok különböző eredménnyel zárulhatnak, ami nem igazán megfelelő megismételhető kísérletek lebonyolítására. A "reduce to subset" és a "reduce to complement" fázisok járják végig a konfigurációkat valamilyen irányban (előrefelé vagy hátrafelé), így ismert, hogy melyik konfigurációt kellene először megvizsgálni. A következő változtatásokat eszközöltük, hogy stabilizáljuk az algoritmust: amennyiben egy reprodukáló tesztesetet találtunk egy tesztelési ablakban, akkor minden még folyamatban lévő tesztvégrehajtást meg kell várni (nem hagyunk el eredményeket). Amennyiben többet találtunk, akkor a bejárási irány alapján válasszuk ki azt, amelyikkel tovább folytatjuk a redukciót.

Ha több reprodukáló tesztesetet találunk és az algoritmus választ egyet a bejárás alapján, a többi konfiguráció eredménye el lesz dobva, még akkor is, ha hasznosak lettek volna. Ez felesleges tesztvégrehajtásokat eredményez olyan konfigurációkon, amik már le lettek tesztelve (és reprodukálták a hibát). A következő stratégia segít csökkenteni a tesztvégrehajtások számát: Ha egy tesztelési ablakban több reprodukáló konfigurációt találtunk, megéri megpróbálni összekombinálni őket és megnézni vajon a kombinált konfiguráció is reprodukál-e. Ameny-nyiben igen, számos tesztvégrehajtást megspórolunk egy lépésben. Ellenkező esetben a bejárási irány alapján válasszuk ki az elsőt és egyenként próbáljuk összekombinálni a többivel. Még ez a megközelítés is spórolhat, mivel csak a reprodukáló konfigurációk különféle kombinációit vizsgáljuk újra és nem egy teljes tesztelési ablakot a következő párhuzamos ciklusban.

Bemutattuk a módosított párhuzamos DDMIN-t (GreeDDy-nek neveztük el) és kiértékel-tük egy publikusan elérhető adathalmaz részhalmazán. Azt találtuk, hogy a GreeDDy a tesztvégrehajtások 31%-át meg tudja spórolni, ami 40%-kal kevesebb futásidőt jelent a DDMIN*-hoz viszonyítva.

A fejezet kontribúciói:

1. A DDMIN* stabilitási problémáinak ismeretetése és egy stabilizálási megoldás bemutatása.

2. A DDMIN* párhuzamos végrehajtását gyorsabbá tétele, kihasználva a lehetőségeket a már látott tesztelési ablakokban (GreeDDy).

3. Több reprodukáló teszteset összevonásával további tesztvégrehajtások spórolhatók meg, és kiderült, hogy a *GreeDDy** a tesztvégrehajtások 31%-át meg tudja spórolni a DDMIN*-nek, ami 40%-al gyorsabb redukciót eredményezett. A *GreeDDy** eredményéről általánosságban elmondható, hogy kisebb lett, viszont az algoritmus hatása a kimenetre elhanyagolható.

Az eredmények a 6. Fejezetben vannak bemutatva, melyek a [36] tanulmányra épülnek.

## 4. Extending Hierarchical Delta Debugging with Hoisting

Habár a HDD és a variánsai jobban teljesítenek struktúrával rendelkező teszteseteken, mint a DDMIN, mindig van esély némi optimalizációra. Számos javítás lett már publikálva, sokszor a HDD által feldolgozott fa adatszerkezetén végeznek előfeldolgozást, pl. elrejtenek néhány csomópontot a HDD elől így kevesebb tesztvégrehajtást eredményezve. Másik előfeldolgozás lehet a csomópontok összeolvasztása ugyanezen okból, vagy a rekurzív struktúrák forgatása a fa magasságának csökkentése érdekében. Viszont, ezek a transzformációk nem változtatják meg a fa alapvető struktúráját, az abból előállítható teszteset ugyanaz lesz, mint az előfeldolgozás előtt.

Vannak visszatérő struktúrák HDD bemenetét képző fa adatszerkezetben, amiket a HDD nem képes redukálni, viszont egy szakértő egyszerűen kiszúrna,

mint pl. feltételes utasítások, ciklusok, függvényhívások paraméterlistán belül stb. Ezt a problémakört megoldván, a transzformáció-alapú minimalizálás egy algoritmikus keretrendszert ír le, amely nem csak törlés, hanem áthelyezés alapú transzformációkat is megenged. Egy ilyen transzformációt definiáltunk is (amit “emelés” -nek (Hoist) neveztünk), ami azt feltételezi, hogy egy részfa helyettesíthető egy másikkal a szintaktikus helyesség elvesztése nélkül, ha a két részfa gyökerei ős-leszármazott viszonyban vannak és a két gyökérelem ugyanolyan címkéjű.
A fejezet kontribúciói:

1. A transzformáció-alapú minimalizálás keretrendszerének formalizálása, továbbá egy példa transzformáció, az emelés. Az emelés mögötti gondolat az, hogy egy részfa kicserélhető egy másik részfával a szintaktikai helyesség elvesztése nélkül, ha a két részfa gyökere azonos címkéjű és ős-leszármazott viszonyban vannak.

2. Az emelés kiértékelése különböző konfigurációban. Először az emelés a HDD előfeldol-gozási lépéseként volt alkalmazva (pl. Hoist* + HDD*), majd egymásba ágyazva a HDD-vel (pl. HDDH*) végül ez a két lépés nem zárja ki egymást, sorozatban mindkettő elvégezhető (pl. Hoist* + HDDH*).

3. Valós adatokon az emelés kombinálva a HDD-vel általánosságban kisebb kimenetet eredményezett, de legalább akkorát, mint amit a HDD egyedül elérne. Nagyobb kimenetek ritkák. A redukált tesztesetek akár a tradicionális HDD kimenetének egyötöde is lehetnek.

4. Az emelés hatása a HDD-re és a HDD$^r$-re (rekurzív HDD) hasonló: a tesztesetek többsége tovább csökkenthető az emelés hatására.

5. A Coarse HDD és a Coarse HDD$^r$ hasonló mintázatot mutat a kimeneti méret te-kintetében: a tesztesetek tovább csökkenthetők az emelés alkalmazásával. Viszont az emelésnek nincs hatása a Coarse HDDH és a Coarse HDDH$^r$ algoritmus variánsokra, illetve, amikor az emelés előfeldolgozási lépésként volt alkalmazva, akkor az algoritmusok viselkedése nem változott meg.

6. Az emelés hatása a redukció hatékonyságára nagymértékben függ a fa adatstruktúra magasságától. Amennyiben a fa kicsi (kb. 50 magas), akkor az emelés növekményt okozott a tesztvégrehajtások számában, viszont, ha a fa elég magas (nagyobb, mint 150), akkor a tesztesetek gyorsabban redukálhatók az emelés alkalmazásával.

A Hoist* + HDDH* és Hoist$^{r}$* + HDDH$^{r}$* algoritmus variánsok állították elő a legkisebb kimenetet a teszteltek közül és a Coarse variánsok igényelték a legkevesebb tesztvégrehajtást (cserébe nagyobb kimenetet produkáltak).

Az eredmények a 7. Fejezetben vannak bemutatva, melyek a [33, 34] tanulmányokra épülnek.

***A szerző hozzájárulásai***

A fejezetekben bemutatott optimalizációk jelentős részének tervezésében, megvalósításában és kiértékelésében a szerzőnek döntő szerepe volt.

1. **Cache Optimizations**: A szerző elemezte a redukcióban alkalmazott korszerű gyorsítótárazási megoldásokat, majd három optimalizálást tervezett és valósított meg a redukció memóriafelhasználásának csökkentése érdekében. Ezt követően kiértékelte a különböző redukciós megközelítésű javaslatok hatásait több teszthalmazon.

2. **Iterating the Minimizing Delta Debugging Algorithm**: A szerző megtervezte és prototípusként elkészítette a DDMIN algoritmus fixpontos iterációját.

3. **Parallel Optimizations of DDMIN***: A szerző elemezte a párhuzamos DDMIN gyengeségeit, majd megtervezte, megvalósította és kiértékelte a megoldást.

4. **Extending Hierarchical Delta Debugging with Hoisting**: A szerző a HDD bemeneteinek (absztrakt szintaxisfák) szerkezetét vizsgálta, optimalizálási lehetőségeket keresve. Megállapította, hogy az azonos címkével ellátott csomópontok a szintaktikai helyesség elvesztése nélkül cserélhetők. Ezután elkészítette a transzformáció-alapú minimalizálási keretrendszer prototípusát, példatranszformációként megvalósította az emelést, majd nyilvánosan elérhető tesztcsomagokon kiértékelte.

Ezenkívül a publikációk replikációs csomagjai minden publikációkor megjelentek. A szerző feladata volt az elkészült algoritmusok nyílt forráskódú publikációja is. A tézisponthoz kapcsolódó publikációk a következők:

[37] **Dániel Vince** and Ákos Kiss. Cache Optimizations for Test Case Reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442-453, Guangzhou, China (Virtual), December 2022. IEEE.

**Table B.1:** *A dolgozat témáinak összefoglalása és a kapcsolódó publikációk*

|   | [37] | [32] | [35] | [36] | [33] | [34] |
|---|---|---|---|---|---|---|
| 1 | ● |  |  |  |  |  |
| 2 |  | ● | ● |  |  |  |
| 3 |  |  |  | ● |  |  |
| 4 |  |  |  |  | ● | ● |

[32] **Dániel Vince**. Iterating the Minimizing Delta Debugging Algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57-60, Singapore, November 2022. ACM.

[35] **Dániel Vince** and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. In *Journal of Software: Evolution and Process*, 2024. Wiley.

[36] **Dániel Vince** and Ákos Kiss. GreeDDy: Accelerate Parallel DDMIN. In *Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '24)*, pages 1-4, Vienna, Austria, September 2024. ACM.

[33] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending Hierarchical Delta Debugging with Hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60-69, Madrid, Spain (Virtual), May 2021. IEEE.

[34] **Dániel Vince**, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of Hierarchical Delta Debugging. In *Journal of Software: Evolution and Process*, 34(11):e2483:1-e2483:26, November 2022. Wiley.

A szerző megjegyzi, hogy bár az ebben a dolgozatban bemutatott eredmények az ő fő hozzájárulását jelentik, a *mi* kifejezést használjuk az *én* helyett önhivatkozásként, hogy elismerjük azon cikkek társszerzőinek hozzájárulását ahhoz, melyek ezen dolgoat alapját szolgáltatják.

# Acknowledgments

Firstly, I would like to thank Dr. Ákos Kiss, my supervisor, for his professional help and unique opinions during my PhD studies. I will not forget my colleagues who gave valuable feedback on the manuscript: Dr. Dombi József Dániel, Zsolt Borbély, and Edit Szűcs. I would like to express my gratitude for the continuous support of my family. Then, I would like to thank Amanda, my wife, who supported me all the way, even though I spent the evenings quietly on the sofa, writing this text. Finally, I am really grateful to Noel, my son, whose arrival put a hard deadline to my writing.

One or more research papers, the result of which were used in this thesis, were partially supported by

- GINOP-2.3.2-15-2016-00037: the EU-supported Hungarian national grant,
- NKFIH-1279-2/2020: of the Ministry for Innovation and Technology, Hungary,
- 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,
- ÚNKP-22-3-SZTE-469 and ÚNKP-23-3-SZTE-536: New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

# Bibliography

[1] Cyrille Artho. Iterative delta debugging. In *Hardware and Software: Verification and Testing*, volume 5394 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2009.

[2] Daniella Bársony. OpenGL API call trace reduction with the minimizing delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 53–56. ACM, November 2022.

[3] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 109–120. ACM, November 2014.

[4] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. ORBS and the limits of static slicing. In *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*, pages 1–10. IEEE, September 2015.

[5] David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. Tree-oriented vs. line-oriented observation-based slicing. In *Proceedings of the 17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2017)*, pages 21–30. IEEE, September 2017.

[6] David Binkley, Nicolas Gold, Syed Islam, Jens Krinke, and Shin Yoo. A comparison of tree- and line-oriented observational slicing. *Empirical Software Engineering*, 24(5):3077–3113, October 2019.

[7] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, pages 1–5. ACM, August 2009.

[8] Nicolas Bruno. Minimizing database repros using language grammars. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, pages 382–393. ACM, March 2010.

[9] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI event traces. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, pages 422–434. ACM, November 2016.

[10] Golnaz Gharachorlu and Nick Sumner. Avoiding the familiar to speed up test case reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 426–437. IEEE, July 2018.

[11] Golnaz Gharachorlu and Nick Sumner. PARDIS: Priority aware test case reduction. In *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019)*, volume 11424 of *Lecture Notes in Computer Science*, pages 409–426. Springer, April 2019.

[12] Nicolas E. Gold, David Binkley, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Generalized observational slicing for tree-represented modelling languages. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pages 547–558. ACM, August 2017.

[13] Ch Anwar Ul Hassan, Muhammad Hammad, Mueen Uddin, Jawaid Iqbal, Jawad Sahi, Saddam Hussain, and Syed Sajid Ullah. Optimizing the performance of data warehouse by query cache mechanism. *IEEE Access*, 10:13472–13480, 2022.

[14] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, pages 861–871. IEEE, October 2017.

[15] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 135–145. ACM, August 2000.

[16] Renáta Hodován. *Fuzz Testing and Test Case Reduction*. PhD thesis, University of Szeged, 2019.

[17] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016)*, pages 31–37. ACM, November 2016.

[18] Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) – Volume 1: ICSOFT-EA*, pages 241–248. SciTePress, July 2016.

[19] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Coarse hierarchical delta debugging. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, pages 194–203. IEEE, September 2017.

[20] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In *Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing (AST 2017)*, pages 23–29. IEEE, May 2017.

[21] Renáta Hodován, Dániel Vince, and Ákos Kiss. Fuzzing javascript environment apis with interdependent function calls. In *Integrated Formal Methods*. Springer International Publishing, 2019.

[22] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Trans. Storage*, 12(2), feb 2016.

[23] Ákos Kiss. Generalizing the split factor of the minimizing delta debugging algorithm. *IEEE Access*, 8:219837–219846, December 2020.

[24] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. HDDr: A recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2018)*, pages 16–22. ACM, November 2018.

[25] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 142–151. ACM, May 2006.

[26] Ghassan Shakib Misherghi. Hierarchical delta debugging. Master's thesis, University of California, Davis, California, June 2007.

[27] Kristi Morton and Nicolas Bruno. FlexMin: A flexible tool for automatic bug isolation in DBMS software. In *Proceedings of the 4th International Workshop on Testing Database Systems (DBTest '11)*, pages 1:1–1:6. ACM, June 2011.

[28] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, pages 291–309. USENIX Association, March 2016.

[29] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. ReduKtor: How we stopped worrying about bugs in Kotlin compiler. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, pages 317–326. IEEE, November 2019.

[30] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE '18)*, pages 361–371. ACM, May 2018.

[31] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[32] Dániel Vince. Iterating the minimizing delta debugging algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST'22)*, pages 57–60. ACM, November 2022.

[33] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. Extending hierarchical delta debugging with hoisting. In *Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021)*, pages 60–69. IEEE, May 2021.

[34] Dániel Vince, Renáta Hodován, Daniella Bársony, and Ákos Kiss. The effect of hoisting on variants of hierarchical delta debugging. *Journal of Software: Evolution and Process*, 34(11):e2483:1–e2483:26, November 2022.

[35] Dániel Vince and Ákos Kiss. Evaluation of the fixed-point iteration of minimizing delta debugging. *Journal of Software: Evolution and Process*, (n/a):e2702.

[36] Dániel Vince and Ákos Kiss. Greeddy: Accelerate parallel ddmin.

[37] Dániel Vince and Ákos Kiss. Cache optimizations for test case reduction. In *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022)*, pages 442–453. IEEE, December 2022.

[38] Shin Yoo, David Binkley, and Roger Eastman. Seeing is slicing: Observation based slicing of picture description languages. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 175–184. IEEE, September 2014.

[39] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, September 1999.

[40] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.