2017 IEEE/ACM 12th International Workshop on Automation of Software Testing

Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Dugonics tér 13, 6720 Szeged, Hungary
Email: hodovan@inf.u-szeged.hu, akiss@inf.u-szeged.hu, gyimothy@inf.u-szeged.hu

***Abstract*—Test case reduction has been automated since the introduction of the minimizing Delta Debugging algorithm, but improving the efficiency of reduction is still the focus of research. This paper focuses on Hierarchical Delta Debugging, already an improvement over the original technique, and describes how its input tree and caching approach can be changed for higher efficiency. The proposed optimizations were evaluated on artificial and real test cases of 6 different input formats, and achieved an average 45% drop in the number of testing steps needed to reach the minimized results – with the best improvement being as high as 82%, giving a more than 5-fold speedup.**

***Index Terms*—Testing, HDD, tree, preprocessing, caching.**

I. Introduction

Programs fail. If we are fortunate enough, they fail deterministically and we can get our hands on an input that reliably induces that failure. We are rarely so lucky though to get a minimal input on the first attempt, one that contains nothing else but what is needed to reproduce the problem. This is when test case reduction has to be applied, preferably in an automated manner.

The seminal paper of Hildebrandt and Zeller [1] showed the way by describing how Delta Debugging (DD) could be applied to test case simplification. Since then, several authors have focused either on adapting the technique to various scenarios [2]–[4], or on improving its efficiency [5], [6]. Even though test case reduction is not manual labor anymore, the time needed to reach the minimal test case can and does matter.

The variant of the technique this paper focuses on is called Hierarchical Delta Debugging (HDD), invented for dealing with the widespread type of hierarchical (tree-represented) inputs [7]. In our previous work [8], we showed how to bring HDD up to speed (input representation and implementation-wise), but during the active use of the technique for fuzzer-generated test case minimization, we have still found ways to make it more efficient. In this paper, we describe two tree preprocessing optimization steps and argue for the use of a caching technique, and we evaluate these ideas on both artificial and real test cases to show how they affect the efficiency of HDD. (The combination of all three approaches achieves a 45% reduction in the number of executed test cases during minimization on average, but can skip as much as 82% in the best case.)

The rest of the paper is organized as follows: first, in Section II, we give a brief overview of Hierarchical Delta Debugging, then in Section III, we describe the proposed improvements, and in Section IV, we present the results of their evaluation. In Section V, we discuss related work. Finally, in Section VI, we give a summary of our work and conclude the paper.

II. Background

The minimizing Delta Debugging algorithm (also called *ddmin*) takes a set, or configuration, representing a failure-inducing test case as input and greedily tries to remove subsets of this configuration while keeping the remaining test case failure-inducing. In the most widespread use case, the elements of a configuration are the lines or characters of a text file and removal means actual deletion of parts from the content. The details of the algorithm are available in papers from Zeller and Hildebrandt [1], [9], [10]. HDD, as first defined in the paper of Misherghi and Su [7], takes Delta Debugging to the domain of hierarchical inputs (e.g., to XML DOM trees or parse trees) by iteratively applying *ddmin* to levels of trees, progressing downwards from the root to the leaves. In that case, configurations are sets of tree nodes and removal of a node causes the removal of the whole subtree rooted at that node. This approach can produce less invalid test cases than a structure-unaware DD, thus result in fewer test attempts and faster minimization. It has to be noted that this HDD algorithm has no guarantees on minimality in general, however, the fixed-point iteration of the algorithm, called HDD*, is shown to yield 1-tree-minimal results. For the sake of completeness, we give the pseudocode of HDD in Listing 1 (with some small changes in the presentation compared to the original formulation to keep its style in sync with the rest of this paper). The auxiliary routines *prune* and *tagNodes* are trivial, so we do not explain them here – the reader is to rely on the original paper.

Although HDD is defined on arbitrary trees, even the first paper considered trees built by using context-free grammars as the general, ideal case. Moreover, it was soon realized that complete subtree removal was not necessarily the best pruning approach for several input formats, as so-pruned trees could easily generate syntactically invalid test cases and therefore cause many superfluous test attempts. Thus, in a subsequent work [11], Misherghi introduced the concept of the smallest allowable syntactic fragments which should be

978-1-5386-1548-5/17 $31.00 © 2017 IEEE
DOI 10.1109/AST.2017.4

Listing 1. The Hierarchical Delta Debugging Algorithm

```
1  procedure HDD(tree)
2  begin
3    level := 0
4    nodes := tagNodes(tree, level)
5    while nodes ≠ ∅ do begin
6      minconfig := DDMIN(nodes)
7      prune(tree, level, minconfig)
8      level := level + 1
9      nodes := tagNodes(tree, level)
10   end
11 end
```

used as replacements for nodes that are not kept by the Delta Debugging algorithm. A recursive formula, Φ, was also given to compute these replacements from context-free grammar rules.

In their work, Misherghi and Su identified another problem with context-free grammars, namely that they use recursion to deal with lists, which in turn yields unbalanced trees unfit for HDD. The first mentioned idea to tackle this issue was to use grammar annotations, but in our previous work [8], we have argued for the use of extended context-free grammars, which gave a solution with the help of well-established theoretical and practical tools.

III. Improvements

Even with the use of extended context-free grammars, there remain cases where HDD performs suboptimally. We will use a running example of manageable size throughout this section to highlight some issues and help in discussing our proposed improvements.

In our example, a fictional program takes a JSON [12] file as input but has some restrictions on how data has to be organized: it must contain exactly one array of single-digit non-zero integers. However, we assume that someone has made an implementation mistake and instead of strictly checking the input against these restrictions only JSON syntax is validated and that numbers in the array fall between 1 and 9, inclusively. We also assume that this sloppyness, i.e., that floating-point numbers can slip through input validation, will cause a failure in our program. The corresponding testing function – using the notations from Zeller et al. – is given in Figure 1a, while a failure-inducing input is shown in Figure 1b.

Since we want to use Hierarchical Delta Debugging to find a reduced test case that still reproduces the failure, we will first use a grammar-based parser to build a tree from the input and compute the smallest allowable syntactic fragments for each node. In Figure 1c, we give a trimmed-down (but still valid and useful-for-the-purpose) version of the JSON grammar in ANTLR v4 format [13], in Figure 1d, we present the calculated minimal replacements, and in Figure 1e, we show the parse tree built from our input.

The results of running HDD on our example, both the output tree and the re-generated JSON representation, are shown in Figures 1f and 1g. Note that the first element of the output array (`2`) is not necessary to reproduce the failure but the structure of the grammar (the "value (`','` value)*" part), and the minimal replacement of token `','` being itself do not allow it to be erased completely. Moreover, it cannot be replaced with Φ[NUM] = `'0'` either as the resulting test case would not be failure-inducing anymore (even the "sloppy" input validation would detect that one of the array elements is out of bounds). Thus, although the output is not minimal byte or token-wise, it is 1-tree-minimal [7]. The removal of that component is not in the scope of this paper.

A. Squeezing the Tree Vertically

As is visible even from our small example, parse trees can contain linear components: paths where each node has at most one child. In the example, the *json–value–array* subgraph and all the *value–NUM* subgraphs are such components. Intuition tells us that such linear components cause unnecessary test attempts: if HDD already visited the topmost node of a linear component at a given level and DD decided to keep it, then

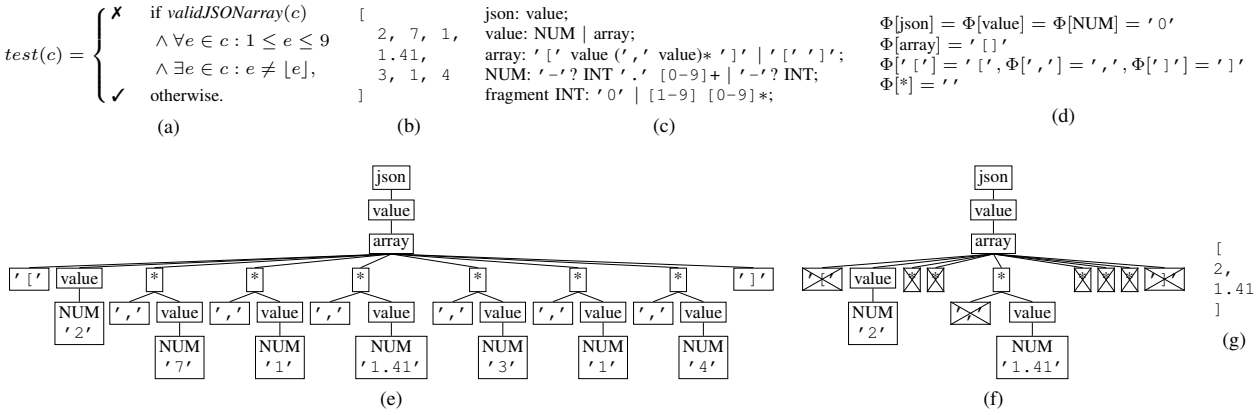

Fig. 1. An HDD example: (a) a testing function, (b) a failure-inducing JSON input, (c) a simplified JSON grammar, (d) minimal replacement strings calculated from the grammar, (e) parse tree built from the input using the grammar, (f) the tree after HDD (nodes marked for removal crossed out), and (g) the corresponding JSON output.

Listing 2. The Squeeze Tree Algorithm

```
1  procedure squeezeTree(node)
2  begin
3    if not isToken(node) then begin
4      forall i in 1..children(node) do
5        child(node, i) := squeezeTree(child(node, i))
6      if children(node) = 1 and Φ(node) = Φ(child(node, 1)) then
7        return child(node, 1)
8    end
9    return node
10 end
```

Listing 3. The Hide Unremovable Tokens Algorithm

```
1  procedure hideUnremovableTokens(node)
2  begin
3    if isToken(node) then
4      if text(node) = Φ(node) then
5        markAsRemoved(node)
6    else
7      forall i in 1..children(node) do
8        hideUnremovableTokens(child(node, i))
9  end
```

it will not decide otherwise for the rest of the nodes of the subgraph either, as it progresses to the next levels. If the minimal replacement strings of the nodes are identical, then the opposite is true as well, i.e., marking any node in the subgraph as removed will yield the same output. Thus, collapsing such lines into a single node can squeeze the height of the tree and reduce the number of tests.

The optimization, which can be applied to the root of the tree before HDD is actually invoked, is formalized as algorithm *squeezeTree* in Listing 2. The auxiliary predicate *isToken* holds for nodes representing tokens (or terminals) from the grammar, *children* gives the number of children of internal nodes, while *child* selects one of those children.

Figure 2 shows the effect of the algorithm on our JSON example. In the tree, only paths of length 2 are collapsible (*json–value* and *value–NUM*, but not *value–array* as Φ[value] $\neq$ Φ[array]). However, in experiments with other inputs and grammars, we have seen even 18 node long linear paths squeezed (more on that in Section IV). Also note that no JSON output is listed because the output is identical to what is shown in Figure 1g.

B. Hiding Unremovable Tokens from DDMIN

Another interesting thing we can notice in our example is that even though some of the nodes are marked as removed in the HDD-reduced tree (those crossed out), they show up in the output nevertheless. The reason behind it is that the smallest allowable syntactic replacement computed for some tokens from the grammar is identical to the occurrence of the token in the input. This is especially true for the anonymous tokens, like `'['`, `','`, and `']'` in the example, but it can occur in general, too. The consequence is that no matter how DD decides, whether to keep them (in which case the actual token text will be used) or not (in which case the replacement will contribute to the output), it will yield the same test case. But DD will try to keep and remove them anyway, leading to superfluous test attempts.

However, if we already marked such "unremovable" tokens as removed in a preprocessing step, before applying HDD to the tree, then they would not be collected by *tagNodes*, effectively hiding them from *ddmin*. The optimization is formalized as the algorithm *hideUnremovableTokens* in Listing 3. The auxiliary function *text* gives the textual representation of a token as it appeared in the input, while *markAsRemoved* is the same as used during HDD, i.e., instead of completely removing a node, it replaces it with the computed syntactic fragment (and prevents it from being tagged ever again). The effect of the algorithm on the parse tree of the example is shown in Figure 3. (The tree after HDD is not shown again as it is identical to the tree of Figure 1f. But the decrease we expect is not in the size of the output but in the number of test attempts anyway.)

C. Content-based Caching

Delta Debugging, the basis of HDD, has configuration of changes as its key concept. The changes may be the lexical decompositions of an input, but in general, they can stand for other notions of changes as well. Because of this generality, although DD implementations do apply caching of test outcomes, these outcomes are usually (actually, in all DD implementations known and available to us) associated with configurations. Furthermore, HDD is not only defined based on Delta Debugging and on the configurations of the nodes of a tree level, but is often also implemented with a generic DD variant. However, while configuration-based caching can be good enough for generic Delta Debugging, it may not be the best approach for HDD: various configurations of tree nodes at a given level may induce the same output, configurations of tree nodes on different levels may induce the same output, and configurations of different HDD* iterations may also induce the same output – which means that all such configurations will yield the same test outcome, too. Unfortunately, if the caching of outcomes is done on the basis of the tree nodes of a given level, none of these recurrences can be detected, i.e., the outcomes will not be retrieved from the cache but will require repeated test attempts.

In Figure 4, we show two examples of node configurations from different tree levels yielding the same outputs (all of them taken from the actual execution of HDD on our running example). In Figures 4a and 4c, nodes from different levels constitute the configuration, they even determine distinct trees, but still generate the same passing (i.e., invalid but not failure-inducing) output. Similarly, in Figures 4d and 4f, distinct trees determined by node configurations from two different levels cause the generation of the same failing output.

While configuration-based caching cannot prevent the re-testing of such trees, associating outcomes with the content of the output does remove such redundancies. Therefore, we propose to optimize HDD by using content-based caching instead of the traditional configuration-based one.

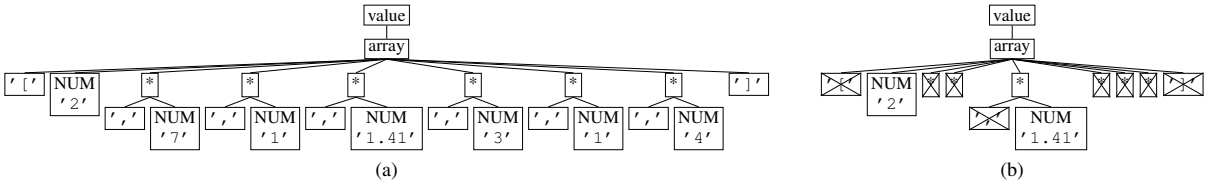

Fig. 2. The effect of the squeeze tree algorithm: (a) the tree of Figure 1e after squeezing and (b) the squeezed tree after HDD.

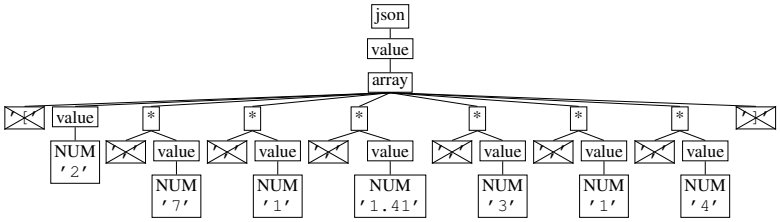

Fig. 3. The effect of the hide unremovable tokens algorithm: the tree of Figure 1e after the algorithm.

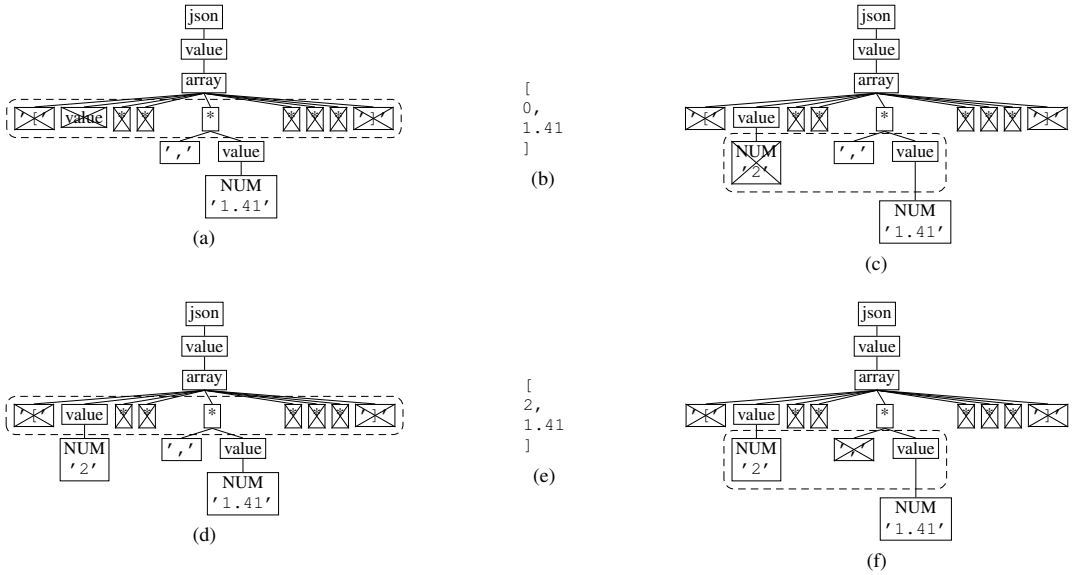

Fig. 4. Various configurations tested by HDD during the minimization of the example of Figure 1: (a) a passing configuration of the 4th level, (c) a passing configuration of the 5th level, (b) the output of both passing configurations, (d) a failing configuration of the 4th level, (f) a failing configuration of the 5th level, and (e) the output of both failing configurations.

IV. Experimental Results

To evaluate the aforementioned ideas in practice, we implemented them in the 17.1 releases of the *Picire*[1] and *Picireny*[2] projects. Since caching fundamentally belongs to the *ddmin* algorithm, the new content-based approach was placed in *Picire*. On the other hand, trees are HDD-related concepts and, accordingly, the tree preprocessing steps went into *Picireny*.

The experiments were executed on a platform with an Intel Core i7 CPU clocked at 2200 MHz equipped with 16GB RAM. The machine was running macOS Sierra 10.12.2 with kernel Darwin 16.3.0.

For the sake of comparability, our experiments start with re-evaluating test cases from prior work [7], [8]. These consist of a Java, a C, and an HTML source, and also a test case with the mixture of HTML, CSS, and JavaScript. The Java source contains a division by zero, the C example discovered a crash in GCC in version 2.95.2, while the HTML-only and the mixed content-type tests caused assertion failures in the rendering engine of Google Chrome (Blink) at revision 402879. Additionally, we used our running JSON example from Section III, and a real life HTML+JavaScript source from GitHub's website that caused an assertion failure in WebKit –

[1]https://github.com/renatahodovan/picire
[2]https://github.com/renatahodovan/picireny

TABLE I
BASELINE

Test	Input File & Tree				HDD (1 / *)			Output File & Tree (1 / *)			
	Size	Height	Rules	Tokens	Tests	Cache Hits	Iter	Size	Height	Rules	Tokens
JSON	33 B	6	16	15	26 / 34	25 / 29	2	11 B	6	6	2
Java	18804 B	35	8821	5937	63 / 80	0	2	37 B	18	17	0
C	672 B	75	1264	278	647 / 1545	381 / 803	3	82 B / 81 B	75	199 / 195	12
HTML	31173 B	101	10254	6493	353 / 453	57 / 65	2	238 B	96	97	3
HTML & CSS & JS	29610 B	57	9411	5751	1142 / 1881	977 / 1299	3	321 B / 294 B	34	151 / 127	17 / 15
HTML & JS	880882 B	123	364541	260117	17058 / 23187	187041 / 189554	5	3805 B / 582 B	52 / 48	1762 / 317	293 / 54

TABLE II
EFFECT OF SQUEEZE TREE

Test	Optimized Tree		HDD (1 / *)		Output Tree (1 / *)	
	Height	Rules	Tests	Cache Hits	Height	Rules
JSON	4 (67%)	8 (50%)	22 / 27 (85% / 79%)	17 / 19 (68% / 66%)	4 (67%)	3 (50%)
Java	28 (80%)	4683 (53%)	60 / 74 (95% / 93%)	0 (–)	15 (83%)	14 (82%)
C	32 (43%)	307 (24%)	303 / 558 (47% / 36%)	216 / 354 (57% / 44%)	25 (33%)	53 / 51 (27% / 26%)
HTML	73 (72%)	7101 (69%)	324 / 395 (92% / 87%)	53 / 57 (93% / 88%)	69 (72%)	68 (70%)
HTML & CSS & JS	50 (88%)	6675 (70%)	1021 / 1608 (89% / 85%)	959 / 1205 (98% / 93%)	29 (85%)	116 / 99 (77% / 78%)
HTML & JS	107 (87%)	248222 (68%)	15459 / 21995 (91% / 95%)	148747 / 151383 (80% / 80%)	45 / 43 (87% / 90%)	1489 / 240 (85% / 76%)

TABLE III
EFFECT OF HIDE UNREMOVABLE TOKENS

Test	Optimized Tree	HDD (1 / *)	
	Tokens	Tests	Cache Hits
JSON	7 (47%)	20 / 28 (82% / 80%)	17 / 21 (68% / 72%)
Java	1749 (29%)	58 / 75 (94% / 92%)	0 (–)
C	78 (28%)	514 / 1412 (79% / 91%)	264 / 686 (69% / 85%)
HTML	2781 (43%)	343 / 443 (97% / 98%)	56 / 64 (98% / 98%)
HTML & CSS & JS	2165 (38%)	917 / 1656 (80% / 88%)	676 / 998 (69% / 77%)
HTML & JS	91122 (35%)	13207 / 19467 (77% / 84%)	49990 / 52538 (27% / 28%)

TABLE IV
EFFECT OF CONTENT-BASED CACHE

Test	HDD (1 / *)	
	Tests	Cache Hits
JSON	19 (73% / 56%)	32 / 44 (133% / 151%)
Java	55 (87% / 69%)	8 / 25 (– / –)
C	280 / 489 (43% / 32%)	748 / 1859 (196% / 232%)
HTML	241 / 242 (68% / 53%)	169 / 276 (296% / 425%)
HTML & CSS & JS	868 / 1181 (76% / 63%)	1251 / 1999 (128% / 154%)
HTML & JS	12737 / 17534 (75% / 76%)	191362 / 195207 (102% / 103%)

the rendering engine of Apple Safari – at revision 199605. The necessary grammar files were downloaded from the official ANTLR v4 grammar repository[3].

Table I shows the details of these test cases and their reduction with all optimizations disabled. The presented information can be sorted into three categories: the first section contains the size metrics of the input, like the size in bytes, the number of rule and token nodes in the initial HDD trees, and the height of these trees. The second block describes the steps of the HDD and HDD* algorithms including the executed test cases, the cache hits, and the iteration count of HDD*. Lastly, symmetrically to the first section, we give the size metrics of the minimized output. The subsequent three tables show the effect of the individual optimizations, but will only contain those columns that have changed compared to these baseline values. The output file sizes are missing from all tables since every version resulted in exactly the same output.

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

Table II shows the results of the minimization of the test cases after applying the vertical squeezing step on the HDD trees. Since this preprocessing step only removes inner nodes, i.e., rule nodes, columns with token counts are not repeated. As the table shows, the amount of rule nodes and also the height of trees decreased significantly. After this step, the trees had 30–76% fewer internal nodes and their height decreased by 12–57%. (The longest squeezed lines were 2 for the JSON test case, 5 for Java, 18 for C, and 3 for all HTML-containing tests. The outstanding results of the C test case are attributed to the highly chained nature of the expression productions in the available C grammar.) However, this decrease of rule nodes is not directly commensurate with the drop in the number of executed test cases: removing nodes is only effective if they are in a kept subtree. On the other hand, removing inner nodes changes the structure of the tree and so the content of the certain levels, which alters the result of DD on that level. In general, we can say that we could decrease the number of

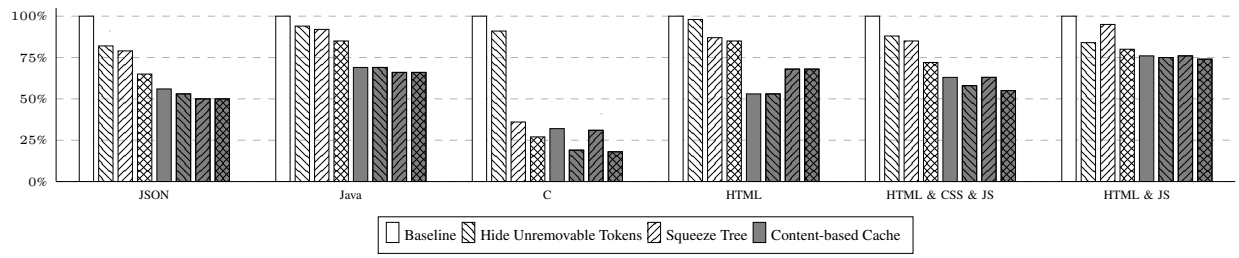

Fig. 5. Relative number of test cases executed by HDD*

TABLE V
OVERALL IMPROVEMENT

Test	HDD (1 / *)	
	Tests	Cache Hits
JSON	17 /17 (65% / 50%)	15 / 22 (60% / 76%)
Java	53 / 53 (84% / 66%)	1 / 15 (– / –)
C	163 / 281 (25% / 18%)	97 / 372 (25% / 46%)
HTML	308 / 309 (87% / 68%)	58 / 132 (102% / 203%)
HTML & CSS & JS	755 / 1037 (66% / 55%)	603 / 1154 (62% / 89%)
HTML & JS	11608 / 17253 (68% / 74%)	49600 / 53127 (27% / 28%)

needed test cases by at least 5% and with a maximum of 64%.

Table III shows what happens when the unremovable tokens are hidden from *ddmin*. This change cannot alter the number of rule nodes; it has an effect on the number of token nodes only and hence the steps and cache hits needed for reducing. As the first column shows, 53–72% of the tokens are completely useless since their unparsed representation is exactly the same both in kept and removed states. However, as was also experienced with tree squeezing this remarkable reduction of participating nodes does not correlate explicitly with the number of executed test cases, because *ddmin* behaves differently on the changed configuration. In this case, the gain we achieved in the test runs is 2% minimum and 23% at most.

Table IV shows the outcome of investigating content-based caching. This change does not alter the tree at all, so its influence is confined to the number of run test cases and cache hits. According to the results, content-based caching is a powerful optimization: it caused a 13–68% decrease in the number of test runs thanks to using 2–325% more outcomes from cache.

After evaluating the impact of the optimizations separately, we combined them to measure how they worked together. Since our primary goal is to decrease the number of needed test executions, we present this information in Figure 5. The shade and pattern of the bars denote the applied optimizations. The first white bars serve as a baseline that marks the unoptimized implementation. If a bar's pattern contains north-west lines, then the unremovable tokens are hidden. The presence of north-east lines denotes that tree squeezing was applied. If a bar has a gray background, then we used content-based caching while running HDD*. As the previous tables already revealed, hiding nodes and squeezing the tree decreases the amount of executed test cases but their efficiency strongly depends on the structure of the tree and the place of the failure-inducing subtree. The bars also show that if we apply only one of the optimizations, then content-based caching is the most powerful. However, if we take a look at the combined results, then we get a bit more nuanced view. When token hiding and tree squeezing are applied together, their effects on the number of spared tests approximately add up. On the other hand, when we pair them with content-based caching, then the gain is not so obvious, if there is any. This behavior arises from the fact that all of the optimizations intend to eliminate duplicate test runs, either by squeezing the input tree, or by hiding the unremovable tokens, or by explicitly caching the executed test cases. However, while tree-squeezing and token-hiding usually eliminate different kinds of recurrences, content-caching easily overlaps with both of them diminishing their efficiency.

In Table V, we summarize the overall gain achieved when applying all optimizations. As the table shows, we could spare at least 13% of the executed tests, but depending on the test case, this ratio could even reach 82%.

V. RELATED WORK

Since its debut, Delta Debugging [1], [9], [10] has become the basis for many works that aim at localizing those parts of an input which are responsible for a certain behavior. Although its original purpose was to identify the failure inducing-parts of text-based inputs, nowadays it is used in many other scenarios, too, e.g., for simplifying failure inducing UI events [4] or for aiding observation-based slicing [14]. However, most current research still intends to lower the number of needed test executions.

One of the early findings is that the character and line-based splitting of the original approach is inefficient, since it can break strongly dependent language constructs, resulting in many syntax errors. These test cases cannot reproduce the original issue, so executing them means overhead on the reduction. The first attempt to eliminate this problem was *Delta*[4] that

[4]http://delta.tigris.org

used the *topformflat* tool. It can heuristically reformat the input by flattening various nesting levels – bounded by opening and closing braces – into one line, hence decreasing the number of syntactically incorrect tests during line-based reduction.

From the same motivation, Misherghi and Su proposed the idea of Hierarchical Delta Debugging [7], [11] where they used context-free grammars to parse the input and build tree representation from it. Using this tree, they ran Delta Debugging on the levels of trees. In our previous work [8], we suggested using extended context-free grammars – that could have regular expressions in the right-hand side of parser rules – for the tree building step of HDD. With this modification, the initial tree became much more balanced and, due to the better handling of quantified subexpressions, HDD resulted in smaller outputs with fewer executed tests.

But not only HDD attempted to exploit knowledge about the syntax of the input. Bruno developed SIMP [15], an approach for minimizing reproductions of database-related problems that uses grammars both for parsing queries and also for analyzing the built AST to identify syntactically interchangeable subtrees. This information was used to build a reduction strategy that could efficiently replace them with each other.

As another example of language-dependent approaches, Regehr et al. introduced four Delta Debugging-based algorithms [16] to reduce C files causing bugs in GCC and they could achieve a 25 times reduction speedup over other public tools.

Contrary to the previous examples, there is still room to improve Delta Debugging even without additional knowledge about the input. In previous work [6], we proved that the implicit sequentiality in the original DD algorithm was not needed to ensure 1-minimality. Leveraging this finding, we could define a parallel alternative to DD, along with other variants that could significantly improve its performance.

VI. Summary

In this paper, we have presented three optimization ideas to decrease the number of test executions during Hierarchical Delta Debugging, thereby improving its efficiency: the *squeeze tree* and the *hide unremovable tokens* algorithms, and the idea of *content-based caching*. The techniques have been implemented in publicly available projects and evaluated on a set of test cases featuring six different input formats (JSON, Java, C, HTML, CSS, and JavaScript). Separately, they could achieve 64%, 23%, and 68% reduction in the testing steps, respectively, but when combined, this went up to 82% in the best case, equating to a more than 5-fold speedup of the minimization process.

Hitherto, our works mainly focused on improving Delta and Hierarchical Delta Debugging while keeping their minimality guarantees intact. As for the future, we still plan to keep our focus on DD and HDD but also on variants that trade some portion of theoretical minimality for practical performance gains. Additonally, we would like to investigate techniques for the simplification of hierarchical inputs that are not DD or HDD anymore but apply more free-form tree transformations.

Acknowledgment

This research was partially supported by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037.

References

[1] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. ACM, Aug. 2000, pp. 135–145.

[2] A. Vida, "Random test case generation and delta debugging for bit-vector logic with arrays," Master's thesis, Johannes Kepler University Linz, Oct. 2008.

[3] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: Delta debugging, even without bugs," *Software Testing, Verification & Reliability*, vol. 26, no. 1, pp. 40–68, Jan. 2016.

[4] A. Elyasov, W. Prasetya, J. Hage, and A. Nikas, "Reduce first, debug later," in *Proceedings of the 9th International Workshop on Automation of Software Test*. ACM, 2014, pp. 57–63.

[5] J. Regehr, "Parallelizing delta debugging," Jul. 2011, http://blog.regehr.org/archives/749.

[6] R. Hodován and Á. 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*. SciTePress, Jul. 2016, pp. 241–248.

[7] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, May 2006, pp. 142–151.

[8] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2016)*. ACM, Nov. 2016, pp. 31–37.

[9] A. 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)*, ser. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Sep. 1999, pp. 253–267.

[10] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[11] G. S. Misherghi, "Hierarchical delta debugging," Master's thesis, University of California, Davis, Jun. 2007.

[12] *The JSON Data Interchange Format (ECMA-404)*, 1st ed., Ecma International, Oct. 2013.

[13] T. Parr, *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2013.

[14] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 109–120.

[15] N. Bruno, "Minimizing database repros using language grammars," in *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*. ACM, Mar. 2010, pp. 382–393.

[16] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, Jun. 2012, pp. 335–346.