Practical Improvements to the Minimizing Delta Debugging Algorithm

Renáta Hodován and Ákos Kiss
Department of Software Engineering, University of Szeged, Dugonics tér 13, 6720, Szeged, Hungary

Keywords: Test Case Minimization, Delta Debugging, Parallelization.

Abstract: The first step in dealing with an input that triggers a fault in a software is simplifying it as much and as automatically as possible, since a minimal test case can ensure the efficient use of human developer resources. The well-known minimizing Delta Debugging algorithm is widely used for automated test case simplification but even its last published revision is more than a decade old. Thus, in this paper, we investigate how it performs nowadays, especially (but not exclusively) focusing on its parallelization potential. We present new improvement ideas, give algorithm variants formally and in pseudo-code, and evaluate them in a large-scale experiment of more than 1000 test minimization sessions featuring real test cases. Our results show that with the help of the proposed improvements, Delta Debugging needs only one-fourth to one-fifth of the original execution time to reach 1-minimal results.

1 INTRODUCTION

Triggering faults in a software is one (undoubtedly important) thing, but once one is triggered, someone has to find and understand its root cause before the correction could even be considered. And the time and effort of that someone are precious. Assuming that the fault is deterministically triggered by a given input, the first step to ensure the efficient use of developer resources is to simplify the input by removing those parts that don't contribute to the failure. This is especially true for those inputs, which are not only large but also hard to comprehend by a human software engineer, e.g., the results of fuzzing or some other random test generation (Takanen et al., 2008). Of course, the more this simplification process can be automated the better. The more than a decade old idea of Delta Debugging from Zeller and Hildebrandt (Zeller, 1999; Hildebrandt and Zeller, 2000; Zeller and Hildebrandt, 2002) is oft-cited and used for exactly this purpose.

Unfortunately, we found that all available generic – i.e., target language-independent – implementations of delta debugging (e.g., the one incorporated into HDD (Misherghi and Su, 2006), the clean-room implementation of the Lithium tool[1], and even the reference implementation provided by Zeller[2]) realize the idea as a sequential algorithm. As multi-core and even multi-processor computers are widespread nowadays, this looked like a suboptimal approach.

This first impression has led us to take a closer look at the original formalization from Zeller and Hildebrandt: we have been looking for ways of improving the performance of the algorithm. Interestingly, parallelization was not the only opportunity for enhancement.

In this paper, we present algorithm variants, which can result as small (i.e., 1-minimal) test cases as those produced by the original formalization but can use resources considerably better. We also present a large-scale experiment with the algorithm variants to prove their usefulness in practice.

The rest of the paper is structured as follows: Section 2 gives a brief overview of the minimizing Delta Debugging algorithm and related concepts. Section 3 discusses our observations on the original algorithm and presents improved variants. Section 4 introduces the setup of our experiments and details the results achieved with a prototype tool implementing all algorithms. Section 5 surveys related work, and finally, Section 6 concludes the paper.

2 A BRIEF OVERVIEW OF DDMIN

In order to make the present paper self-contained, we give Zeller and Hildebrandt's latest formulation of

[1]http://www.squarefree.com/lithium/
[2]http://www.st.cs.uni-saarland.de/dd/DD.py

Hodován, R. and Kiss, Á.
Practical Improvements to the Minimizing Delta Debugging Algorithm.
DOI: 10.5220/0005988602410248
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 241-248
ISBN: 978-989-758-194-6

Algorithm 1: Zeller and Hildebrandt's.

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 Δ_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 $test(c'_{\text{✗}}) = \text{✗} \wedge n \leq |c'_{\text{✗}}|$.

Delta Debugging (Zeller and Hildebrandt, 2002) verbatim in Algorithm 1.

The algorithm takes a test case and a testing function as parameters. The test case represents the failure-inducing input to be minimized, while the testing function is used to determine whether an arbitrary test case triggers the original failure (by signaling *fail* outcome, or ✗) or not (by signaling *pass* outcome, or ✓). (Additionally, according to its definition, a testing function may also signal *unresolved* outcome, or **?**, but that outcome type is irrelevant to the algorithm.)

Informally, the algorithm works by partitioning the input test case to subsets and removing as many subsets as greedily as possible such that the remaining test case still induces the original failure. The algorithm and the partitioning is iterated until the subsets become elementary and no more subsets can be removed. The result of the algorithm is a 1-minimal test case, where the definition of n-minimality is:

Definition 1 (n-minimal test case)**.** *A test case* $c \subseteq c_{\text{✗}}$ *is* n-minimal *if* $\forall c' \subset c \cdot |c| - |c'| \leq n \Rightarrow (test(c') \neq \text{✗})$ *holds. Consequently, c is 1-*minimal *if* $\forall \delta_i \in c \cdot test\,(c - \{\delta_i\}) \neq \text{✗}$ *holds.*

Although the original definition of the algorithm is more of a recursive math formula, all known implementations realize it as a sequential non-recursive procedure, as we already mentioned in Section 1. Therefore, we give an equivalent variant of ddmin$_2$ – the key component of the minimizing Delta Debugging algorithm – in a sequential pseudo-code in Algorithm 2.

Algorithm 2: Non-recursive sequential pseudo-code.

```
procedure ddmin$_2^{\sigma}$($c'_{✗}$, n)
begin
  out:
  while true do begin
    (∗ reduce to subset ∗)
    forall i in 1..n do
      if test($\Delta_i^{(c'_{✗},n)}$) = ✗ then begin
        $c'_{✗}$ = $\Delta_i^{(c'_{✗},n)}$
        n = 2
        continue out
      end
    (∗ reduce to complement ∗)
    forall i in 1..n do
      if test($\nabla_i^{(c'_{✗},n)}$) = ✗ then begin
        $c'_{✗}$ = $\nabla_i^{(c'_{✗},n)}$
        n = max(n − 1, 2)
        continue out
      end
    (∗ increase granularity ∗)
    if n < $|c'_{✗}|$ then begin
      n = min($|c'_{✗}|$, 2n)
      continue out
    end
    (∗ done ∗)
    break out
  end
  return $c'_{✗}$
end
```

3 PRACTICAL IMPROVEMENTS TO DDMIN

If we take a closer look at the minimizing Delta Debugging algorithm, as introduced in the previous section, we can make several observations that can lead to practical improvements. In the next subsections we discuss such improvement possibilities.

3.1 Parallelization

The first thing to notice is that although the implementations tend to use sequential loops to realize the “reduce to subset” and “reduce to complement” cases of ddmin$_2$, as exemplified in Algorithm 2, the potential for parallelization is there in the original formulation, since $\exists i \in \{1,\ldots,n\}$ does not specify how to find that existing i. Since n can grow big for real inputs and *test* is often expected to be an expensive operation, we propose to make use of the parallelization potential and rewrite ddmin$_2$ to use parallel loops. The pseudo-code of the parallelized version is given in Algorithm 3.

In the pseudo-code we intentionally do not specify the implementation details of the parallel constructs but we assume the following for correctness and maximum efficiency:

Algorithm 3: Parallel pseudo-code.

```
1   procedure ddmin$_2^{\pi}$($c'_{✗}$, n)
2   begin
3     while true do begin
4       (∗ reduce to subset ∗)
5       found = 0
6       parallel forall i in 1..n do
7         if $test(\Delta_i^{(c'_{✗},n)})$ = ✗ then begin
8           found = i
9           parallel break
10        end
11      if found ≠ 0 then begin
12        $c'_{✗} = \Delta_{found}^{(c'_{✗},n)}$
13        n = 2
14        continue
15      end
16      (∗ reduce to complement ∗)
17      found = 0
18      parallel forall i in 1..n do
19        if $test(\nabla_i^{(c'_{✗},n)})$ = ✗ then begin
20          found = i
21          parallel break
22        end
23      if found ≠ 0 then begin
24        $c'_{✗} = \nabla_{found}^{(c'_{✗},n)}$
25        n = max(n − 1, 2)
26        continue
27      end
28      (∗ increase granularity ∗)
29      if n < |$c'_{✗}$| then begin
30        n = min(|$c'_{✗}$|, 2n)
31        continue
32      end
33      (∗ done ∗)
34      break
35    end
36    return $c'_{✗}$
37  end
```

- First of all, assignments to a single variable are expected to be atomic. I.e., if found = i gets parallelly executed in two different loop bodies (with values i1 and i2) then the value of found is expected to become strictly either i1 or i2. (See lines 8 and 20.)
- The amount of parallelization in **parallel forall** is left for the implementation, but it is expected not to overload the system, i.e., start parallel loop bodies only until all computation cores are exhausted. (See lines 6 and 18.)
- **parallel break** is suggested to stop/abort all parallelly started loop bodies even if their computation hasn't finished yet (in a general case, this may cause computation results to be thrown away, variables left uninitialized, etc., which is always to be considered and taken care of, but it causes no issues in this algorithm). (See lines 9 and 21.)

3.2 Combination of the Reduce Cases

To improve the algorithm further, we have to take a look again at the original formulation of ddmin$_2$ in Algorithm 1. We can observe that although ddmin$_2$ seems to be given with a piecewise definition, the

Algorithm 4: Parallel pseudo-code with combined reduce cases.

```
procedure ddmin$_2^{\kappa}$($c'_{✗}$, n)
begin
  while true do begin
    (∗ reduce to subset or complement ∗)
    found = 0
    parallel forall i in 1..2n do
      if 1 ≤ i ≤ n then
        if $test(\Delta_i^{(c'_{✗},n)})$ = ✗ then begin
          found = i
          parallel break
        end
      else if n + 1 ≤ i ≤ 2n then
        if $test(\nabla_{i-n}^{(c'_{✗},n)})$ = ✗ then begin
          found = i
          parallel break
        end
    if 1 ≤ found ≤ n then begin
      $c'_{✗} = \Delta_{found}^{(c'_{✗},n)}$
      n = 2
      continue
    end else if n + 1 ≤ found ≤ 2n then begin
      $c'_{✗} = \nabla_{found-n}^{(c'_{✗},n)}$
      n = max(n − 1, 2)
      continue
    end
    (∗ increase granularity ∗)
    if n < |$c'_{✗}$| then begin
      n = min(|$c'_{✗}$|, 2n)
      continue
    end
    (∗ done ∗)
    break
  end
  return $c'_{✗}$
end
```

pieces are actually not independent but are to be considered one after the other, as mandated by the *else if* phrases. However, we can also observe that this sequentiality is *not* necessary. There may be several Δ_i and ∇_i test cases that induce the original failure, we may choose any of them (i.e., we don't have to prefer subsets over complements), and we will still reach a 1-minimal solution at the end.

We cannot benefit from this observation as long as our implementation is sequential but we propose to combine the two reduce cases, and test all subsets and complements in one step when parallelization is available. This way the algorithm does not have to wait until all subset tests finish but can start testing the complements as soon as computation cores become available.

Algorithm 4 shows the pseudo-code of the algorithm variant with the combined reduce cases.

The pseudo-code in earlier sections – i.e., Algorithms 2 and 3 – are consistent with the original formal definition of Zeller and Hildebrandt, as shown in Algorithm 1. They specialize the formalization and differ from each other only in the way how the existency check $\exists i \in \{1,\ldots,n\}$ is implemented (i.e., sequentially or parallelly). However, the idea of combining the reduce cases as presented in Algorithm 4, although still yields 1-minimal results, deviates from the original formalization.

3.3 De-prioritization or Omission of "Reduce to Subset"

Once we have observed that there is no strict need for sequentially investigating the two reduce cases of ddmin_2, we can also observe that the "reduce to subset" case is not even necessary for 1-minimality. It is a greedy attempt by the algorithm to achieve a significant reduction of the test case by removing all but one subsets in one step rather than removing them one by one in the "reduce to complement" case. However, there are several input formats where attempting to keep just the "middle" of a test case almost always gives a syntactically invalid input and thus cannot induce the original failure. (C-like source files with the need for correct file and function headers and mandating properly paired curly braces are a typical example of such inputs.) For such input formats, the "reduce to complement" case may occur significantly more often, while the "reduce to subset" case perhaps not at all. Thus, we argue that it is worth experimenting with the reordering of the reduce cases, and also with the complete omission of the "reduce to subset" case, as it may be simply the waste of computation resources.

The idea of reordering the reduce cases can be applied to all previously introduced algorithm variants. However, presenting the modified algorithms again would be unnecessarily verbose for little benefit. Thus, we only refer to the algorithms shown in previous sections and mention those parts that need to be changed to test *complements first*:

- in Algorithm 2, lines 5–11 and 12–18 should be swapped,
- in Algorithm 3, swapping lines 4–15 and 16–27 achieves the same, while
- in Algorithm 4, lines 8–11 have to be swapped with lines 13–16, lines 18–20 with lines 22–24, and the subscript indices of Δs and ∇s have to be updated so that the $-n$ element is applied to Δs.

The idea of omitting the "reduce to subset" case completely can reasonably be applied to the sequential and parallel variants only, as leaving the testing of subsets from the algorithm with combined reduce cases would be no different from the parallel variant. Thus, the changes to be applied for testing *complements only* are as follows:

- in Algorithm 2, lines 5–11 are to be deleted, while
- in Algorithm 3, lines 4–15 are superfluous.

4 EXPERIMENTAL RESULTS

During the investigation of the minimizing Delta Debugging algorithm, we created a prototype tool[3] that implemented our proposed improvements. At the very beginning, it was based on Zeller's public domain reference implementation but as new ideas got incorporated into it, it was rewritten several times until only traces of the original source remained. The tool was written in Python 3, and the parallel loop constructs of the algorithm variants were implemented based on Python's mutiprocessing module.

For the evaluation platform of our tool and our ideas, we have used a dual-socket Supermicro X9DRG-QF machine equiped with 2 Intel Xeon E5-2695 v2 (x86-64) CPUs clocked at 2.40 GHz and 64 GB DDR3 RAM at 1600 MHz. Each CPU had 12 cores and each core was hyper-threaded, which gave 24 physical cores and 48 processing threads (or logical CPUs seen by the kernel) in total. The machine was running Ubuntu 14.04 with Linux kernel 3.13.0, and the native compiler was gcc 4.9.2.

As primary software targets to trigger failure in and minimize tests for (also known as system under test, or SUT), we have chosen two real-life web engines – from the WebKit[4] and Chromium[5] projects –, for which a large number of test cases were available. The WebKit project was checked out from its official code repository at revision 192323 dated 2015-11-11 and built in debug configuration for its GTK+ port, i.e., external dependencies like UI elements were provided by the GTK+ project[6]. The SUT was the *WebKitTestRunner* program (executed with the *--no-timeout* command line option), a minimalistic web browser application used in the testing of layout correctness of the project. The Chromium project was checked out from its official code repository at revision hash 6958b6e dated 2015-11-29 and also built in debug configuration. The SUT from that project was the *content_shell* tool (executed with the *--single-process --no-sandbox --run-layout-test* options), also a minimal web browser. As a secondary target, we have implemented a mock SUT, a parametrized tester function that can unit test the algorithm implementations for correctness by working on abstract arrays as inputs and allowing an explicit control of what is accepted as failure-inducing or passing, and also allows experimenting with various test execution times (since the time required for the evaluation of the failure condition is negligible and the tester can lengthen

[3]https://github.com/renatahodovan/picire
[4]https://webkit.org/
[5]https://www.chromium.org/
[6]http://www.gtk.org/

Table 1: Test cases, sizes, and their interestingness condition.

Test Case	Size	Condition
Chromium A (html)	31,626 bytes / 1,058 lines	ASSERTION FAILED: i < size()
Chromium B (html)	34,323 bytes / 3,769 lines	ASSERTION FAILED: static_cast<unsigned>(offsetInNode) <= layoutTextFragment−>start() + layoutTextFragment−>fragmentLength()
Chromium C (html)	48,503 bytes / 1,706 lines	ASSERTION FAILED: !current.value()−>isInheritedValue()
WebKit A (html)	23,364 bytes / 959 lines	ASSERTION FAILED: newLogicalTop >= logicalTop
WebKit B (html)	30,417 bytes / 1,374 lines	ASSERTION FAILED: willBeComposited == needsToBeComposited(layer)
WebKit C (html)	36,051 bytes / 3,791 lines	ASSERTION FAILED: !needsStyleRecalc() || !document().childNeedsStyleRecalc()
Example A (array)	8 elements	$\{5,8\} \subseteq c \wedge (2 \in c \vee 7 \notin c)$
Example B (array)	8 elements	$\{1,2,3,4,5,6,7,8\} \subseteq c$
Example C (array)	8 elements	$\{1,2,3,4,6,8\} \subseteq c \wedge \{5,7\} \nsubseteq c$
Example D (array)	100 elements	$\{2x \vert 0 \leq x < 50\} \subseteq c$

its running time arbitrarily).

For each of the two browser engine targets, we have selected 3 fuzzer-generated test cases – HTML with a mixture of SVG, CSS, and JavaScript – that triggered various assertion failures in the code. The average size of the tests was around 2000 lines, with the shortest test case being 959 lines long and the longest one growing up to 3769 lines. For the mock SUT, we have used 4 manually crafted test cases, which consisted of an array of numbers and a condition to decide about the interestingness of a certain subset. Three of them were small tests imported from Zeller's reference examples, while the larger fourth (with 100 elements) was specially crafted by us for a corner case when reduction becomes possible only when granularity is increased to the maximum. Table 1 details the sizes and the interestingness conditions of all test cases.

In the evaluation of the algorithm variants that we have proposed in the previous sections, our first step was to check the sizes of the minimized test cases. The reasons for the investigation were two-fold: first, as the manually crafted test cases have exactly one 1-minimal subset, they acted as a sanity check. Second, as 1-minimal test cases are not necessarily unique in general, we wanted to see how the algorithm variants affect the size of the result in practice, i.e., on the real browser test cases.

We have examined the effect of the reduce step variations ("subsets first", "complements first", and "complements only"; as described in Section 3.3) on the algorithm and the effect of the two parallelization approaches ("parallel" and "combined"; as described in Sections 3.1 and 3.2) independently. It turned out that all reduce variants of the sequential algorithm gave exactly the same result not only for the examples tests but for the real browser cases as well. Detailed investigations have revealed that "reduce to subset" is a very rare step in practice (it happened in the first iteration only, with $n = 2$, when subsets and complements are equal anyway) and because of the sequential nature of the algorithm, the "reduce to complement" steps were taken in the same order by all algorithm variants. Parallel variants – executed with the loops limited to 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, and 64 parallel bodies – did not show a big impact on the result size either, although 0–3 lines of differences appeared. (Clearly, there were multiple failing complements in some iterations and because of the parallelism, the algorithms did not always choose the same step as the sequential variant.) Since those 0–3 lines mean less than 1% deviation in relative terms, we can conclude that none of the algorithm variants deteriorate the results significantly. As a last step of this first experiment, we have evaluated all algorithm variants as well (i.e., parallel algorithms with reduce step variations), and their results are summarized in Table 2. The numbers show that not only did not get the results significantly worse, but for WebKit B parallelization variants found a smaller 1-minimal test case than the sequential algorithm.

In our second experiment, we have investigated the effect of the algorithm variants on the number of iterations and test evaluations needed to reach 1-minimality. In Table 3(a), we can observe the effects of the variation of the reduce steps in the sequential algorithm on all test cases. Although the number of iterations never changed, the number of *test* evaluations dropped considerably everywhere. Even if we used caching[7] of test results, the "complements first" variant saved 12–17% of *test* evaluations on real browser test cases, while "complements only" saved 35–40%. The test cases of the mock SUT show a bit more scattered results with the "complements first" variant (0–23% reduction in test evaluations), but "complements only" achieves a similar 36–41%. As test results are re-used heavily by the original "subsets first" approach, if we would also interpret those cache hits as test evaluations then that could mean a dramatic 90% reduction in some cases. Table 3(b) contains the results of parallelizations – yet again with 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, and 64 parallel loop bodies allowed. These results might look disappointing or

[7]In the context of Delta Debugging, caching is the memorization and reuse of the outcomes of the testing function on already investigated test cases. The caching mechanism is not incorporated in the definition of the algorithm, since it can be left for the *test* function to realize.

Table 2: Size of 1-minimal test cases.

Test Case	Sequential	Parallel	Combined
Chromium A	31 (2.93%)	31–34 (2.93–3.21%)	31–34 (2.93–3.21%)
Chromium B	34 (0.90%)	34–37 (0.90–0.98%)	34–37 (0.90–0.98%)
Chromium C	9 (0.53%)	9 (0.53%)	9 (0.53%)
WebKit A	11 (1.15%)	11 (1.15%)	11 (1.15%)
WebKit B	21 (1.53%)	19–23 (1.38–1.67%)	19–23 (1.38–1.67%)
WebKit C	46 (1.21%)	46 (1.21%)	46 (1.21%)
Example A	2 (25.00%)	2 (25.00%)	2 (25.00%)
Example B	8 (100.00%)	8 (100.00%)	8 (100.00%)
Example C	6 (75.00%)	6 (75.00%)	6 (75.00%)
Example D	50 (50.00%)	50 (50.00%)	50 (50.00%)

Table 3: The number of iterations and test evaluations needed to reach 1-minimality. First numbers or intervals in each column stand for evaluated tests, the second numbers or intervals in parentheses with + prefix stand for cache hits, while the third numbers or intervals in brackets stand for iterations performed.

(a) Comparison on the effects of reduce step variations (performing all tests sequentially).

Test Case	Subsets First	Complements First	Complements Only
Chromium A	531 (+1589) [76]	466 (+54) [76]	343 (+24) [76]
Chromium B	560 (+1931) [83]	489 (+9) [83]	356 (+3) [83]
Chromium C	263 (+388) [56]	218 (+5) [56]	159 (+2) [56]
WebKit A	265 (+479) [53]	222 (+7) [53]	161 (+3) [53]
WebKit B	503 (+1674) [84]	430 (+21) [84]	319 (+7) [84]
WebKit C	831 (+5127) [129]	714 (+14) [129]	509 (+5) [129]
Example A	22 (+22) [8]	17 (+5) [8]	14 (+1) [8]
Example B	26 (+2) [3]	26 (+2) [3]	14 (+0) [3]
Example C	30 (+16) [5]	28 (+3) [5]	18 (+1) [5]
Example D	472 (+3237) [57]	422 (+16) [57]	276 (+0) [57]

(b) Comparison on the effects of parallelization (testing subsets first).

Test Case	Sequential	Parallel	Combined
Chromium A	531 (+1589) [76]	531–2476 (+1568–2389) [76–89]	531–2933 (+1574–2386) [76–89]
Chromium B	560 (+1931) [83]	560–2016 (+1614–1936) [76–83]	560–2947 (+468–1932) [76–83]
Chromium C	263 (+388) [56]	263–589 (+388–394) [56]	263–958 (+22–388) [56]
WebKit A	265 (+479) [53]	265–675 (+477–486) [53]	265–708 (+452–483) [53]
WebKit B	503 (+1674) [84]	503–2116 (+1668–1771) [84–85]	503–2132 (+1623–1762) [84–85]
WebKit C	831 (+5127) [129]	831–5670 (+5127–5136) [129]	831–5697 (+5115–5132) [129]
Example A	22 (+22) [8]	22–35 (+22–23) [8]	22–53 (+5–22) [8]
Example B	26 (+2) [3]	26 (+2) [3]	26–28 (+0–2) [3]
Example C	30 (+16) [5]	30–38 (+16) [5]	30–51 (+3–16) [5]
Example D	472 (+3237) [57]	472–3398 (+3237–3251) [57]	472–3440 (+3009–3249) [57]

controversial at first sight as the number of evaluated tests increased, sometimes by a factor of 6 or even more. However, as we will soon see, this increase is the natural result of parallelization and is not mirrored in the running time of the minimizations.

In our third and last experiment, we have investigated the effect of the algorithm variants on the running time of minimization. First, we have taken a closer look at the WebKit A test case. We found that even without parallelization, varying the reduce steps could decrease the running time by 14% and 35%, well aligned with the results presented in Table 3(a). However, when we enabled parallelization (e.g., 64 parallel loop bodies), we could reduce the running time by 73–75%. Unfortunately, by combining parallelization and reduce step variants, we couldn't simply multiply these reduction factors. E.g., with 64-fold parallelization, the effect of "complements first" became a 2–5% gain in running time, and the effect of "complements only" also shrank to 7–15%. Nevertheless, that is still a gain, the combined parallel loops with complements tests performed only gave the best results, and reduced the time of the minimization to 72 seconds from the original 316, achieving an impressive 77% reduction.

However, we did not only focus on the WebKit A test case, but we have taken all 6 real browser tests, plus parametrized the 4 artificial test cases to run for 0 seconds, 1 second, and randomly between 1 and 3 seconds. That gave 18 test cases in practice, for 3×3 algorithm variants, on 12 different levels of parallelization. Even though we skipped the re-evaluation of duplicates (e.g., "complements first" is the same for "parallel" and "combined" variants, or sequential algorithms run unchanged independently of the available computation cores), we got the results of 1098 minimization executions, all of which would be hard to picture within the limits of this paper. Thus, we present only the best results for each test case in Table 4.

The results clearly show that for most of the cases (12 of 18) the "parallel complements only" variant gave the highest running time reduction – as expected. Perhaps it is worth discussing those elements of the table, which are a bit less expected: first of all, it might be surprising that parallelization did not always work best at the highest level used (64). However, we have to recall that the machine used for evaluation has 24 real cores only (and 48 logical CPUs seen by the kernel, but those are the result of hyper-threading), and it should also be considered that the browser SUTs are inherently multi-threaded or even

Table 4: Algorithm variants giving the highest reduction in running time of minimization.

Test Case	Best Algorithm (Parallelization Level)	Best/Original Time	Reduction
Chromium A	Parallel Complements-Only (24)	231s / 904s	74.42%
Chromium B	Parallel Complements-Only (16)	180s / 890s	79.78%
Chromium C	Parallel Complements-Only (8)	102s / 413s	75.28%
WebKit A	Parallel Complements-Only (12)	70s / 316s	77.63%
WebKit B	Parallel Complements-Only (12)	127s / 584s	78.26%
WebKit C	Parallel Complements-Only (12)	192s / 915s	78.97%
Example A (0s)	Sequential Complements-Only (1)	0.0020s / 0.0023s	13.79%
Example A (1s)	Combined Subsets-First (64)	7s / 22s	67.79%
Example A (1–3s)	Combined Subsets-First (32)	11s / 45s	75.34%
Example B (0s)	Sequential Complements-Only (1)	0.0011s / 0.0014s	23.58%
Example B (1s)	Parallel Complements-Only (16)	3s / 26s	88.29%
Example B (1–3s)	Parallel Complements-Only (16)	7s / 50s	85.91%
Example C (0s)	Sequential Complements-Only (1)	0.0016s / 0.0020s	20.80%
Example C (1s)	Parallel Complements-Only (24)	5s / 30s	83.13%
Example C (1–3s)	Parallel Complements-Only (16)	8s / 53s	84.78%
Example D (0s)	Sequential Complements-Only (1)	0.0436s / 0.0762s	42.76%
Example D (1s)	Parallel Complements-Only (64)	58s / 472s	87.69%
Example D (1–3s)	Parallel Complements-Only (64)	79s / 939s	91.48%

multi-process, thus even a single instance may occupy more than one cores. It may also come as a surprise that a sequential algorithm turned out to perform best for some artificial test cases. These were the cases when the evaluation of the interestingness of a test was so quick that the overhead of the parallelization implementation became a burden. (In practice, *test* implementations rarely run so fast.)

As a summary, we can conclude that all proposed variants of the Delta Debugging algorithm yielded a measurable speed-up, with best combinations gaining cca. 75–80% of the running time on real test cases, i.e., needing only one-fourth to one-fifth of the original execution time to reach 1-minimal results. Moreover, on manually crafted test cases, even higher running time reduction could be observed.

5 RELATED WORK

The topic of test case reduction is an almost two decades old research area. Many of the works in this subject are based on Zeller and Hilldebrand's publication (Hildebrandt and Zeller, 2000), which gave a general, language independent and greedy solution for the 1-minimal test case reduction problem. Although the approach works well, but due to its generality it can be sub-optimal in performance and this fact gives room for optimizations or specializations.

Misherghi and Su experimented with replacing the letter and line based splitting with code units determined by language grammars (Misherghi and Su, 2006). Running Delta Debugging on code units could avoid the generation of large amounts of syntactically incorrect test cases that would have been thrown away anyway.

Another approaches used static and dynamic analysis, or slicing to discover semantic relationships in the code under inspection and reduce the search space where the failure is placed (Leitner et al., 2007; Burger and Zeller, 2011). Few years later, Regehr et al. used Delta Debugging as one possible method in their C-Reduce test case reducer tool for C sources. This system contains various source-to-source transformator plugins – line-based Delta Debugging among others – to mimic the steps that a human would have done.

Regehr also experimented with running the plugins of C-Reduce in parallel. His initiative results were promising but, to our best knowledge, he hadn't performed extensive evaluations later. The write-up about this work is available on his blog (Regehr, 2011).

6 SUMMARY

In this paper, we have taken a close look at the minimizing Delta Debugging algorithm. Our original motivation was to enable the algorithm to utilize the parallelization capabilities of multi-core and multi-processor computers, but as we analyzed it, we found further enhancement possibilities.

We have realized the parallelization potential implicitly given in the existency checks of the original definition of the algorithm (although never exploited by its authors), but we have also pointed out that the same original definition unnecessarily prescribed sequentiality elsewhere, for its "reduce to subset" and "reduce to complement" steps. Moreover, we have argued that the "reduce to subset" step of the original algorithm may even be omitted completely without losing its key property of resulting 1-minimal test cases but being more effective for some types of inputs. Each observation has led to an improved algorithm variant, all of which are given in pseudo-code.

Finally, we presented an experiment conducted on 4 artificial test cases and on 2 wide-spread browser engines with 3 real test cases each. All test cases were minimized with all presented algorithm variants and with 12 different levels of parallelization (ranging from single-core execution – i.e., no parallelization – to 64-fold parallelization on a computer with 48 virtual cores). The results of the 1098 successfully executed test case minimizations prove that all presented improvements to the Delta Debugging algorithm achieved performance improvements, with best variants reducing the running time on real test cases significantly, by cca. 75–80%.

For future work, we still see improvement possibilities and research topics. We plan to investigate how the best algorithm variant can be selected for a given hardware and SUT, and which order of subset and complement tests work for different input formats the best.

ACKNOWLEDGMENTS

This work was partially supported by the European Union project "REPARA", project number: 609666.

REFERENCES

Burger, M. and Zeller, A. (2011). Minimizing reproduction of software failures. In *2011 International Symposium on Software Testing and Analysis*, pages 221–231.

Hildebrandt, R. and Zeller, A. (2000). Simplifying failure-inducing input. In *2000 International Symposium on Software Testing and Analysis*, pages 135–145.

Leitner, A., Oriol, M., Zeller, A., Ciupa, I., and Meyer, B. (2007). Efficient unit test case minimization. In *22nd International Conference on Automated Software Engineering*, pages 417–420.

Misherghi, G. and Su, Z. (2006). HDD: Hierarchical delta debugging. In *28th International Conference on Software Engineering*, pages 142–151.

Regehr, J. (2011). Parallelizing delta debugging. http://blog.regehr.org/archives/749.

Takanen, A., DeMott, J., and Miller, C. (2008). *Fuzzing for Software Security Testing and Quality Assurance*. Artech House.

Zeller, A. (1999). Yesterday, my program worked. Today, it does not. Why? In *7th European Software Engineering Conference/7th International Symposium on Foundations of Software Engineering*, pages 253–267.

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.