Cache Optimizations for Test Case Reduction

Dániel Vince* and Ákos Kiss
Department of Software Engineering, University of Szeged, Szeged, Hungary
vinced@inf.u-szeged.hu, akiss@inf.u-szeged.hu
*corresponding author

***Abstract*—Finding the relevant part of failure-inducing inputs is an important first step on the path of debugging. If much of a test case that triggers a bug does not contribute to the actual failure, then the time required to fix the bug can increase considerably. In this paper, we focus on the memory requirements of automatic test case reduction. During minimization, the same test case might be tested multiple times, and determining the outcome of an input may take time, therefore, different caching solutions were proposed to avoid re-testing previously seen inputs. We investigated the caching solutions of *DDMIN* and *HDD*, and found that their scaling is suboptimal. We propose three optimizations for one of the state-of-the-art caching solutions: with the optimizations combined, *DDMIN* requires 96% and *HDD* requires 85% less memory compared to the baseline implementation. Furthermore, as a side effect, the reduction becomes faster by 9.9% with *DDMIN*.**

***Keywords*—cache, delta debugging, hierarchical delta debugging, optimization, test case minimization**

I. Introduction

If there is a chance that something can malfunction, that behavior will happen sooner or later. Although software engineering craftsmanship has undergone great development in recent decades, a wide range of errors can still be found in their handmade products, *i.e.*, from misunderstanding the specifications (or a single feature request) to low-level, programming language specific mistakes, *e.g.*, pointer usage in C. When a program failure happens and we are lucky enough, we have the record of events or inputs that triggered the failure. In this case, a lucky engineer will get the task of fixing the problem, but debugging is not considered the most exciting part of software engineering, especially when done manually.

The first step of fixing the problem is finding the minimal subset of the input that caused the faulty behavior. Luckily, several techniques are available to automate this task, and one of the most well-known of them is the minimizing Delta Debugging algorithm (*DDMIN*) by Zeller and Hildebrandt [1], [2], [3], working on all kinds of inputs without the need for any information about their internal structure. If the input structure is known, however, that knowledge can be utilized to create smaller results faster, therefore, Misherghi and Su have introduced the Hierarchical Delta Debugging (*HDD*) algorithm [4], [5], built on *DDMIN*, that works on tree-structured inputs (*i.e.*, on any input format that has a context-free grammar) and prunes unnecessary subtrees of the structure during reduction. Context-free grammars are readily available for several input formats, which makes *HDD* easily applicable in a variety of scenarios. However, the need for a grammar can act as an obstacle for some users of test case reducers if a grammar is unavailable or maintaining it is not a practical option. In such cases, the structure-unaware nature of *DDMIN* is very useful.

Both of the above algorithms can utilize caches to improve their execution time at the cost of increased memory usage. However, the concept of caching is orthogonal to the reduction algorithms themselves, and studies have mainly focused on the latter. In this paper, we focus on caches and their memory footprint aspects. Our goal is to answer the following research questions:

Research Questions

RQ1. How efficient are the state-of-the-art caching techniques for test case reduction?
RQ2. Can caching be modified to reduce memory usage without compromising the effectiveness of the reduction algorithms?
RQ3. What are the effects of optimizations on *DDMIN* and *HDD*?

Thus, we have studied the caching techniques of state-of-the-art *DDMIN* and *HDD*-based reducers and analyzed their efficiency. Then, we have proposed to optimize the caching in order to require less memory without changing the integrity of the process. Finally, we have evaluated our proposals and found that they can help reduce the memory footprint of the minimization process by as much as 96% (*DDMIN*) and 85% (*HDD*) compared to the publicly available solutions.

The rest of the paper is organized as follows: first, in Section II, we give a brief overview of *DDMIN* and *HDD*, to make this paper self-contained. In Section III, we describe our optimization ideas. In Section IV, we evaluate the effects of the optimizations with the help of a prototype implementation, and we present our experimental results, then, in Section V, we discuss related work. Finally, in Section VI, we summarize our work and conclude the paper.

II. Background

The minimizing Delta Debugging (*DDMIN*) algorithm [1], [2], [3] is a systematic iterative approach for

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

Figure 1. The Minimizing Delta Debugging algorithm [3].

reducing arbitrary input while keeping a predefined property invariant. The input is split into atomic units and represented as a set of them. First, this set of units is split into two roughly equal-sized subsets and both parts are investigated about whether they still have the predefined property of the initial input. If the property is kept in any of the subsets, then the reduction step is considered successful and a new iteration starts with that subset, otherwise, the granularity is refined by doubling the splitting. The subsets of the new splitting are investigated again, as well as their complements, *i.e.*, it is checked whether keeping or removing any of the subsets leads to a smaller, yet interesting test case. Again, if any of the investigated subsets (or their complements) 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 the unit level, when it is proven to have found a so-called 1-minimal result, a local minimum where the removal of any unit from the set would cause the loss of the interesting property.

DDMIN has its roots in the isolation of failure-inducing code changes. An input is composed of elementary changes (or deltas), denoted as δ_i, whence the algorithm got its name. A set of elementary changes is also called a configuration, usually denoted by c. The outcome of a program execution on a specific configuration is determined by a testing function, and it can be either *fail* (also written as ✗) if the current input produced the original behavior, *pass* (also written as ✓) if the test succeeds, or *unresolved* (written as ?) if the result is indeterminate. The initial configuration that triggers the failing outcome 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 interesting property of a test case is not a program failure, most authors, including us, follow the original notation for historical reasons. For the sake of completeness, Figure 1 gives Zeller and Hildebrandt’s latest formulation of the minimizing Delta Debugging algorithm [3].

```
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
```

Figure 2. The Hierarchical Delta Debugging algorithm [4].

If an input that is to be minimized has some mandatory structure over its units, which is typical for inputs to a program, *DDMIN* may work suboptimally. The configuration partitioning during the iterations may be unaligned with the boundaries of the structural elements of the input, leading to incorrectly formatted, thus non-reproducing, and therefore, useless test cases. The goal of the Hierarchical Delta Debugging (*HDD*) algorithm [4] is to avoid such superfluous steps by not testing format-breaking configurations. It works on hierarchical tree-structured input representations (*e.g.*, on parse trees, abstract syntax trees, or XML DOM trees) and applies the minimizing Delta Debugging algorithm to the levels of that tree, progressing downwards from the root to the leaves. The pseudocode formulation of *HDD* as defined by Misherghi and Su [4] is shown in Figure 2. 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 removal of the whole subtree rooted from that node. In a latter variant of *HDD*, "pruning" of a node has been reinterpreted as its replacement with the minimal applicable syntactically correct fragment to reduce the number of test attempts at incorrectly formatted configurations even further [5]. If *HDD* is iterated until a fixed-point is reached, denoted as *HDD**, it gives a 1-tree-minimal result, *i.e.*, if any node is removed from the tree, the newly serialized test case would not be interesting anymore.

Zeller [1] also raised the issue that testing an arbitrary configuration may take time. If the input to be tested is a program in source code form, its recompilation and re-execution could take 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.

It might happen that the same configuration is tested multiple times among the iterations. To avoid running the same test twice, Zeller provided an *outcome caching* mechanism in his reference implementation[1]. The cache is implemented as a tree structure where each node is labelled with a unit of the configuration and an outcome that corresponds with the subconfiguration formed by the units from the current node up to the root of the tree. Consider the following configurations in the cache: $(\langle 1,2\rangle, ✗)$, $(\langle 1,2,3\rangle, ✓)$, and $(\langle 1,4,5\rangle, ✗)$. They can be represented in a tree structure as shown in Figure 3. 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 (✗, ✓, ?). If the algorithm creates a new configuration to test, 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. (Note that 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 labelled with an outcome at that point of the algorithm, but may get an outcome assigned later on. Figure 3 shows $\langle 1,4\rangle$ and $\langle 1\rangle$ as examples of such not-yet-tested configurations.)

While the above-described configuration-based cache is efficient with *DDMIN*, it may not be the best approach for *HDD*. Hodován *et al.* formalized this problem in their study [6]: 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 of different *HDD** iterations may also produce the exact same output. All such configurations yield the same test outcome as well. However, 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

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

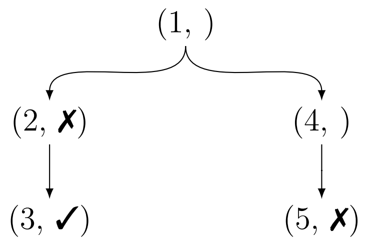


Figure 3. The outcome caching approach of the reference implementation of the Delta Debugging algorithm.

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 can be applied to *DDMIN* as well.

III. Cache Optimizations

Although the content-based cache improved the efficiency of reduction, there is still room for improvement. General purpose caching techniques try to maximize the utilization of the available (fixed size) memory by keeping the most popular entries in cache [7]. The most widespread algorithms for cache replacements are Least Frequently Used (LFU), Most Frequently Used (MFU), and Least Recently Used (LRU) [8], but these classic techniques do not make use of knowledge of the underlying algorithms and thus evict elements from cache that might be needed later. However, in this work, we are interested in reducing the memory footprint of the cache by utilizing the characteristics of the reduction algorithm. Therefore, we propose three optimizations that aim to reduce the memory requirements of the content cache.

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 I shows the character-based reduction process of *DDMIN* step by step. For the sake of simplicity, we skip the "reduce to subset" steps, only the "reduce to complement" operations are presented.

As discussed in Section II, 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 that are larger than n would not be tested, since the new iteration splits that configuration into smaller chunks, *e.g.*, in the 4th step in Table I, 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 notations introduced in Section II:

TABLE I
Execution of 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*	✓

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ \|.\|: \text{size of the serialized configuration} \\ \exists c_x: \textit{test}(c_x) \to \text{✗ found} \\ \forall c_y: \|c_y\| > \|c_x\|: \text{ out of search space} \end{gathered} \tag{1}$$

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, we perform a cache lookup. The cache may contain smaller entries, *e.g.*, in the 6th step in Table I, 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, our *first proposal*, as shown in (2), is to add only *passing* tests to the cache which may reduce the memory footprint of the test case minimization. Furthermore, cache lookups might be quicker since the queries are performed in a smaller search space.

$$\begin{gathered} c_x \subseteq c_{\text{✗}} \\ \text{when } \exists c_x: \textit{test}(c_x) \to \text{✓ found} \\ \textit{insert to cache} \ (\ \textit{serialize}(c_x) \ ) \end{gathered} \tag{2}$$

(The function *insert to cache* inserts an element into the cache, while *serialize* performs the serialization of test cases as discussed in [6].)

Another benefit of (1) is that if a failing test case is found, we can be sure that no cache entry corresponding to larger test cases than the currently found one would be queried during the remaining reduction process. Therefore, when a new failing test case is found, entries that stores the outcome of test cases that is larger than the currently investigated one can be evicted from the cache, as shown in (3). We will refer to this eviction process as our *second proposal.*

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ \text{when } \exists c_x: \textit{test}(c_x) \to \text{✗ found} \\ \forall c_y: \textit{serialize}(c_y) \in \text{cache} \wedge \|c_y\| > \|c_x\|: \\ \textit{delete from cache} \ (\ \textit{serialize}(c_y) \ ) \end{gathered} \tag{3}$$

(The function *delete from cache* implements the removal of an element from the cache.)

For small inputs, this proposal might be a runtime overhead only, however, the benefits might overcome the costs for "large enough" inputs. We assume that cache lookup is quicker than actual test execution, but, of course, it takes time as well. Thus, 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 we follow the above discussed proposals, the cache will contain *passing* tests only and will be cleared after each successful reduction step. However, the lengths of the stored entries are varying, *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 is progressing towards the 1-minimum. Thus, our *third proposal* is the following: the cache should not store the serialized content of the configurations, but their transformed form as shown in (4).

$$\begin{gathered} c_x \subseteq c_{\text{✗}}, M: M \in \mathbb{N} \\ \textit{transform}(c_x): 2^{\mathbb{N}} \mapsto 2^M \text{ bijection} \\ \text{when } \exists c_x: \textit{test}(c_x) \to \text{✓ found} \\ \textit{insert to cache} \ (\textit{transform}(\ \textit{serialize}(c_x) \ )) \end{gathered} \tag{4}$$

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[2]. 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.

$$\begin{gathered} c_x, c_y \subseteq c_{\text{✗}} \\ x: \textit{serialize}(c_x), \ y: \textit{serialize}(c_y) \\ \exists c_x, \exists c_y: x \neq y \implies \\ \textit{transform}(x) = \textit{transform}(y) \end{gathered} \tag{5}$$

Suppose that c_x from (5) has already been tested ($\textit{test}(c_x) \to$ ✓) and inserted into the cache. Now the algorithm tries another configuration c_y ($c_x \neq c_y$), performs

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

a cache lookup, and finds that it has already been tested (since *transform*(x) = *transform*(y)). This state can lead to two different outcomes:

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

test$(c_x) = ✓ \wedge$ *test*$(c_y) = ✗$: c_y would reproduce the initial ✗, but because of the cache hit, it would never be tested.

This may lead to a suboptimal reduction outcome, but the invariants of the algorithm are still not violated. In theory, this may lower the effectiveness of the reduction. Section IV will discuss whether collisions happened in our experiments in practice.

IV. Experimental Results

A. Experiment Setup

To evaluate the effects of the optimizations, we have created a prototype implementation of the proposals based on the open-source Picire[3] and Picireny[4] projects. Picire is a Python implementation of *DDMIN*, supporting parallelization and several configuration options. Picireny is a hierarchical test case reduction framework on top of Picire, also written in Python, that supports ANTLR v4[5] grammars and already contains an implementation of the *HDD* algorithm. We have chosen the *SHA-3-256* algorithm from the Python *hashlib* module for the *transform* function in (4). To measure the cache size during reduction, we have used the *pympler*[6] Python module.

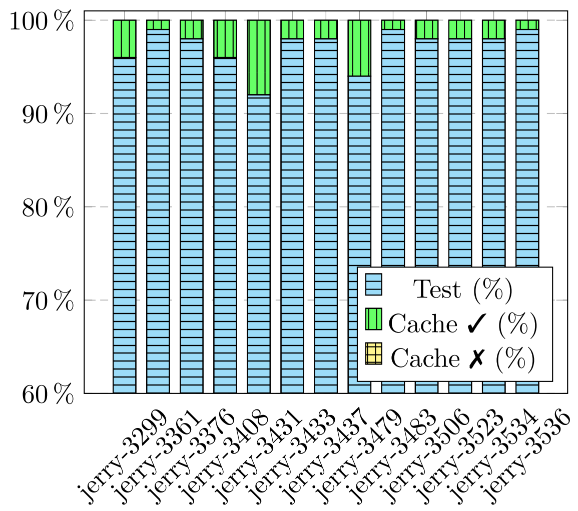
As inputs, we have collected test cases from different sources, all of which have already been used in the literature for benchmarking reduction. The first test suite is the Perses Test Suite[7] (PTS), which contains fuzzer-generated C sources that cause various internal compiler errors in the Clang and GCC compilers. The second test set is the JerryScript Reduction Test Suite[8] (JRTS), which also contains fuzzer-generated JavaScript files that cause failures in the JerryScript lightweight JavaScript engine. In the case of both test suites, the interesting property of the test cases to keep during reduction is the failure they induce. The properties of the test cases are shown in Table II. *Size* is expressed as the number of non-whitespace characters in the test case, *Tree Height* represents the height of the parse tree built from the input, *Rules* show the number of non-terminals, and *Tokens* show the number of terminals in it. The parse tree representation of each test case, to be passed as input to *HDD*, was built using the grammar available for the input format from the official ANTLR v4 grammars repository[9]. Moreover, Picireny has applied the squeezing of linear tree components [6] and the flattening of recursive structures [9] to the trees.

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 we have used tests from JRTS only for benchmarking *DDMIN*. For *HDD*, we have used both test suites. To determine the effects of the proposals on the execution time, we have repeated the experiments multiple times and averaged their execution times. (Note that reduction output and cache behavior 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 5.4.0, and executing the experiments only.

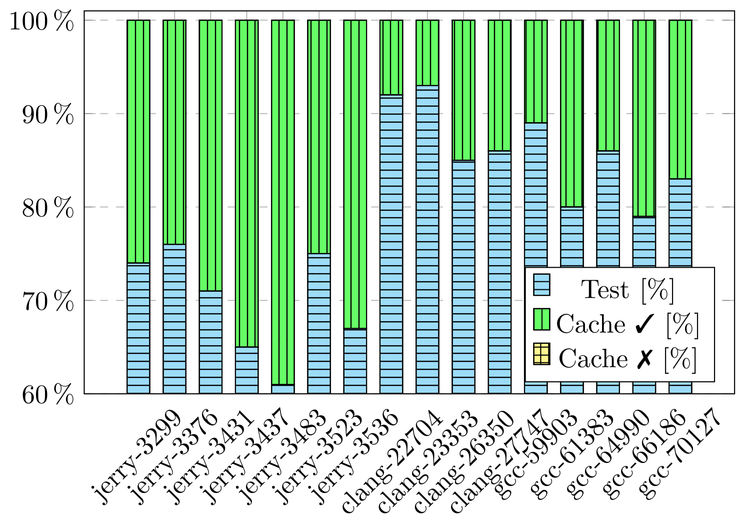
TABLE II
Properties of the Inputs Used for Benchmarking

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

[3]https://github.com/renatahodovan/picire
[4]https://github.com/renatahodovan/picireny
[5]https://github.com/antlr/antlr4
[6]https://pypi.org/project/Pympler/
[7]https://github.com/uw-pluverse/perses
[8]https://github.com/vincedani/jrts
[9]https://github.com/antlr/grammars-v4



(a)



(b)

Figure 4. Cache hits and test executions with (a) *DDMIN* and (b) *HDD*.

B. Efficiency of the Content Cache

In order to make sure optimization ideas presented in Section III 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(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* outcome, while (yellow) bars with grid patterns stand – or, would stand – for *fail* outcomes returned from the cache. Note that, supporting (1), no cache lookup returned a ✗ outcome. Similar observations can be made about *HDD* (see Figure 4(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.

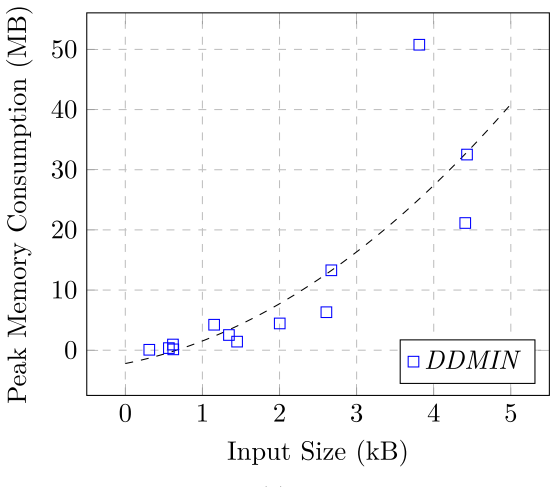
We had one unexpected finding with *HDD* though: 9 cache lookups (0.01% of all cases) returned a ✗ result. After manual analysis of the steps of the algorithm on the test cases we found that this may happen when the minimal replacement of a tree node is identical to its serialized form, *i.e.*, when it does not matter if such a node is pruned or not, the same character sequence would be serialized from it. Therefore, contrary to (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 were 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 5(a) and 5(b). The horizontal axes show the input size (in kB and MB, respectively) and the vertical axes show the peak memory consumption (in MB or GB). Figure 5(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 5(b) shows the same information for the *HDD* algorithm, 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). But with *HDD*, the peak memory consumption could easily reach 4 GB.

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.

C. 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 we compare the effects of the combined 1st and 2nd proposals against the results of the 1st, and finally, we show how all three proposals compare against the combination of the 1st and 2nd. Technically, as described in Section III, we think of our proposals as steps. For *HDD*, the results are presented for all three proposals combined.



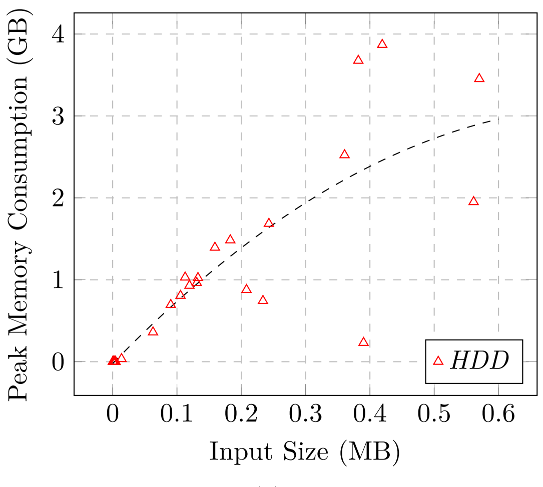


Figure 5. Memory consumption of the content cache with (a) *DDMIN* and (b) *HDD*.

Proposal 1

As motivated in Section III, the first proposal is to avoid adding configurations to the cache that have *failing* outcomes. Figure 6(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 24.29% improvement in the best case. When investigating the execution time, we can see that the change is not consistent: in the best case, 3.45% of the execution time is saved (jerry-3376), but there can also be 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

As (3) shows, our 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 6(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 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. Note that Proposal 1 was considered as the reference in this comparison.

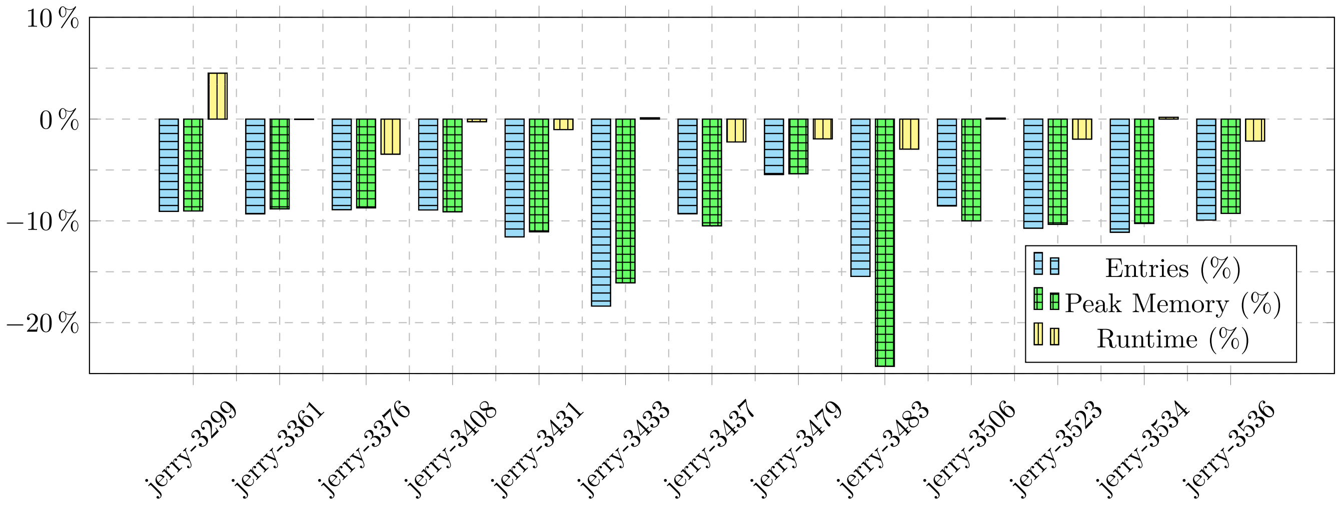
Proposal 3

Our third proposal is about storing a transformed version of the serialized test case in the cache. As mentioned before, we investigated the *SHA-3-256* hashing algorithm 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 I 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-3-256 transformed content of the test case and the value is its size (before the transformation).

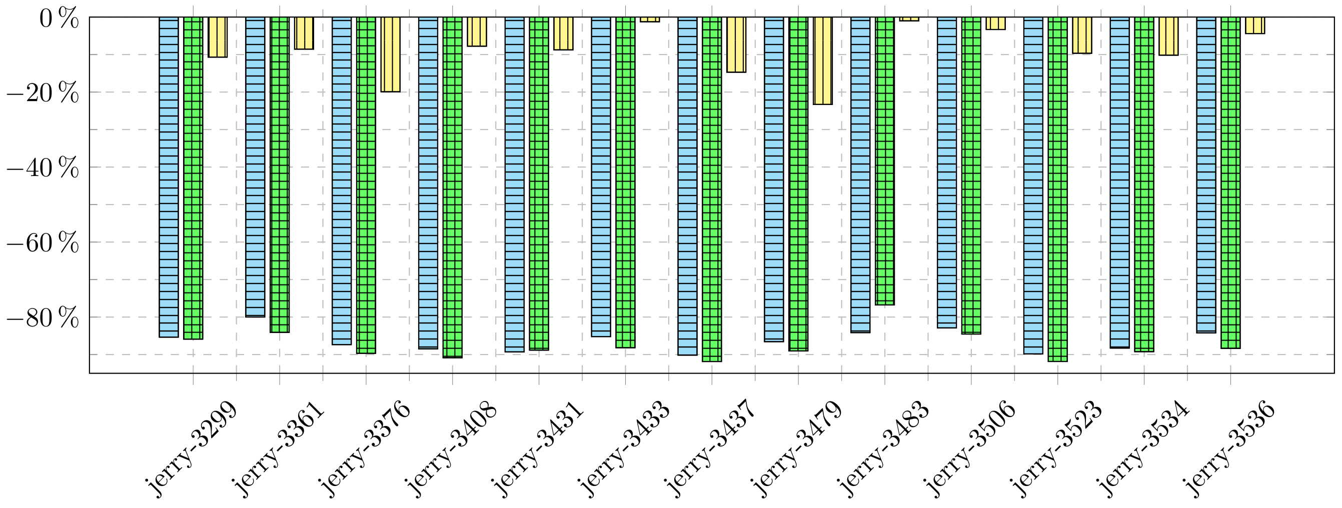
We first examined the question that was raised in Section III, *i.e.*, whether collisions happened during reduction: during our experiments with the used test suites and algorithm implementations, we have not faced any hash collisions at all.

Since only the form of the stored entities changed, Figure 6(c) 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.

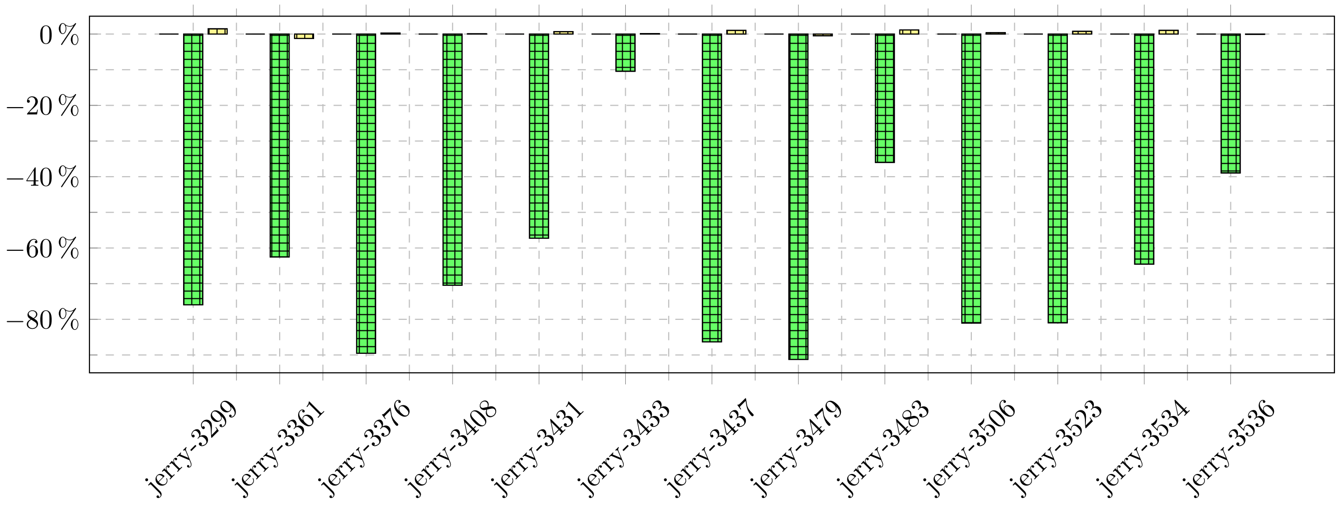
The backing data for the experiments are shown in Table III. The “Baseline” column shows the peak memory



(a)



(b)



(c)

Figure 6. (a) Effects of Proposal 1 on *DDMIN* compared to the baseline [6]. (b) Effects of Proposal 2 on *DDMIN* compared to Proposal 1. (c) Effects of Proposal 3 on *DDMIN* compared to Proposals 1 and 2 combined.

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 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), therefore, we discuss the combined impacts of the optimizations (as the utility is replaced by the *improved DDMIN*).

When investigating experimental results from JRTS, we found that the cache had to store 47.47% fewer entries after enabling all of the optimizations, and this required 63.19% less memory usage on average. We did not see consistent trends 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 99.93% smaller memory footprint and 7.89% shorter runtime. As shown in Table II, the characteristics of the used test suites are quite different and it can be observed that the baseline solution by Hodován *et al.* [6] 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 85% smaller memory footprint in 4.46% shorter time.

The backing data can be found in Table IV, the “Proposals” column shows results after applying all three proposals.

D. Conclusions

Based on the data and observations above, we can conclude the experiments and answer our research questions.

Answer to Research Question #1

How efficient are the state-of-the-art caching techniques for test case reduction?

- Based on our preliminary research, we chose the “content-based” caching technique by Hodován *et al.* [6] as our baseline.
- *DDMIN* determined the outcome of its configurations via the cache only in 3% of the cases.
- *HDD* utilized the cache better, the actual testing of 21% of the configurations could be avoided.
- The baseline caching solution 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.

Answer to Research Question #2

Can caching be modified to reduce memory usage without compromising the efficiency of the reduction algorithms?

In this study, we proposed three optimizations to reduce the memory footprint of caches used in test case minimization:

1) add only *passing* (✓) tests to the cache,
2) when a new *failing* (✗) test case is found, evict cache entries of bigger test cases, and
3) instead of storing the serialized test contents, store their *hashed* value (fixed-width keys instead of variable-width).

In our experiments, the result of the reductions did not change after the optimizations.

Answer to Research Question #3

What are the effects of optimizations on DDMIN and HDD?

- The effects of the optimizations are similar on both algorithms.
- On *DDMIN*, 96% less memory was needed on average.
- On *HDD*, 85% less memory was needed on average. However, 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%.

E. Threats to Validity

We identified the following threats in this study and considered the following actions to avoid or minimize their effects.

Selection of benchmarks: We used two suites of benchmarks, one in C and one in JavaScript programming language, the Perses Test Suite (PTS) and JerryScript Reduction Test Suite (JRTS), respectively. As the formats of the test cases are similar, structured, and come from the same domain of programming languages, the findings may not generalize to all types of test cases. However, we think that the results on these test suites are indicative as they contain real-world test cases and have been used in reduction-related studies [10], [11], [12].

Correctness of implementation: In order to ensure that the implementation of the experiments is correct and

TABLE III
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%*

accurate, we conducted a review of the code. On selected C and JavaScript examples, we traced the behavior of the implementation to validate that it works as intended. Furthermore, the implementation is based on open-source and well-maintained repositories like Picire and Picireny frameworks that have been used in several studies [6], [9], [13], [14], [15], [16], [17], [18], and ANTLR v4.

V. Related Work

One of the first and most well-known works on automated test case reduction is Delta Debugging by Zeller and Hildebrandt [1], [2], [3], minimizing inputs of arbitrary format. The price of its generality is a potentially lowered performance because of format-breaking incorrect test cases generated and evaluated during the reduction process. The authors have 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.

To avoid syntactically broken intermediate test cases, Miserghi and Su proposed to use information about format encoded in context-free grammars, *i.e.*, to convert test cases into a tree representation [4] and apply delta debugging to the levels of the tree. This Hierarchical Delta Debugging approach helped remove parts of the test case that aligned with syntactic unit boundaries. As a further improvement, Miserghi proposed the concept of a syntactically correct replacement for nodes that cannot be completely removed from the test case without causing syntax errors [5]. Hodován *et al.* investigated the performance of the configuration-based outcome cache in their study [6] and found that in different *HDD** iterations or at different tree levels, different node configurations yield the same serialized test case, thus *HDD* tests the same test case multiple times. Based on this finding, they suggested storing the serialized test case in the cache instead of configurations, called content-based cache. We have used this solution as our baseline and optimized it further to consume fewer resources during the reduction.

An interesting analogy between test case reduction and program slicing was recognized by Binkley *et al.* [19], [20], [21], [22], [23], [24]. They have realized that the concepts of slicing (*e.g.*, the program to be sliced or the slicing criterion) can be reformulated as concepts of test case reduction (*e.g.*, the test case or the interestingness property, respectively). Their approach, called observation-based slicing, avoids the complexities of building a dependency graph representation of a program and can work purely at the syntactic level. Stepanov *et al.* [25] suggested a combined approach using program slicing, *HDD*, and Kotlin-specific transformation in their “ReduKtor” prototype tool. They also experienced that during transformations, the tool may encounter configurations that have already been explored. To avoid re-checking, they hashed the AST configurations and stored them together with the outcome. It should be noted though that 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.

The above-mentioned works utilized different caching solutions, but none of the used approaches took the specificities of the reduction algorithms into account. Consequently, none of these works tried to optimize the cache by exploiting such specificities. We have investigated the state-of-the-art reduction tools, and made a step further to maximize the resource efficiency of the reduction.

TABLE IV
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%*

VI. Summary

In this paper, we have been focusing on the memory requirements of test case reduction. We have investigated *DDMIN* and *HDD*, and how they perform on various test suites. We have found that different tools use different strategies, however, their cores are similar: they find a property of the configuration that fully represents that and store that in the cache. The "content-based" solution was selected to improve upon, which works with both algorithms in a way that serializes the input for the SUT from the configuration and stores that as a key beside the outcome whether passed or failed. Based on its open-source implementation, we have prototyped our proposals as follows:

Proposals

1) Store entries in the cache with a *passing* (✓) outcome only.
2) When a new *failing* (✗) test case is found, evict entries of bigger test cases.
3) Instead of storing the test contents in the cache, store their *transformed*, fixed-length representation (SHA-256-3 in our experiments).

In our experiments, the result of the reductions has not changed, but *DDMIN* and *HDD* required 96% and 85% less memory, respectively.

As for future work, we have plans to continue this topic of research in various ways. We wish to conduct further experiments to investigate how different properties of the input (*e.g.*, size, format, structure, *etc.*) affect the results. This includes the investigation of the current test suites as well as potential additional datasets to see whether the current results can be generalized. We would also like to see the implementation of these proposals in more state-of-the-art test case reduction tools, and measure and compare their effect across algorithms. Furthermore, we would like to explore additional caching techniques in order to be able to minimize inputs that do not have a traditional serialized form.

Acknowledgment

This research was supported by project TKP2021-NVA-09. Project no. TKP2021-NVA-09 has been 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. Dániel Vince was supported by the ÚNKP-22-3-SZTE-469 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

References

[1] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *Proc. 7th Eur. Softw. Eng. Conf. Held Jointly With 7th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (ESEC/FSE)*, ser. Lecture Notes in Comput. Sci., vol. 1687. Springer, 1999, pp. 253–267.

[2] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proc. 2000 ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*. ACM, 2000, pp. 135–145.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[4] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*. ACM, 2006, pp. 142–151.

[5] G. S. Misherghi, "Hierarchical delta debugging," Master's thesis, Univ. California, Davis, California, 2007.

[6] R. Hodován, Á. Kiss, and T. Gyimóthy, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *Proc. 12th IEEE/ACM Int. Workshop Automat. Softw. Test. (AST)*. IEEE, 2017, pp. 23–29.

[7] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, “Improving flash-based disk cache with lazy adaptive replacement,” *ACM Trans. Storage*, vol. 12, no. 2, feb 2016.

[8] C. A. U. Hassan, M. Hammad, M. Uddin, J. Iqbal, J. Sahi, S. Hussain, and S. S. Ullah, “Optimizing the performance of data warehouse by query cache mechanism,” *IEEE Access*, vol. 10, pp. 13 472–13 480, 2022.

[9] R. Hodován, Á. Kiss, and T. Gyimóthy, “Coarse hierarchical delta debugging,” in *Proc. 33rd IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*. IEEE, 2017, pp. 194–203.

[10] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, “Perses: Syntax-guided program reduction,” in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*. ACM, 2018, pp. 361–371.

[11] G. Gharachorlu and N. Sumner, “PARDIS: Priority aware test case reduction,” in *Proc. 22nd Int. Conf. Fundam. Approaches Softw. Eng. (FASE)*, ser. Lecture Notes in Comput. Sci., vol. 11424. Springer, 2019, pp. 409–426.

[12] Á. Kiss, “Generalizing the split factor of the minimizing delta debugging algorithm,” *IEEE Access*, vol. 8, pp. 219 837–219 846, Dec. 2020.

[13] R. Hodován and Á. Kiss, “Practical improvements to the minimizing delta debugging algorithm,” in *Proc. 11th Int. Joint Conf. Softw. Technol. (ICSOFT)*, vol. 1. SciTePress, 2016, pp. 241–248.

[14] Á. Kiss, R. Hodován, and T. Gyimóthy, “HDDr: A recursive variant of the hierarchical delta debugging algorithm,” in *Proc. 9th ACM SIGSOFT Int. Workshop Automating Test Case Design, Selection, Eval. (A-TEST)*. ACM, 2018, pp. 16–22.

[15] D. Vince, R. Hodován, D. Bársony, and Á. Kiss, “Extending hierarchical delta debugging with hoisting,” in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 60–69.

[16] D. Vince, R. Hodován, and Á. Kiss, “Reduction-assisted fault localization: Don’t throw away the by-products!” in *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*. SciTePress, Jul. 2021, pp. 196–206.

[17] D. Vince, R. Hodován, D. Bársony, and Á. Kiss, “The effect of hoisting on variants of hierarchical delta debugging,” *Journal of Software: Evolution and Process*, vol. 34, no. 11, p. e2483, 2022.

[18] D. Vince, “Iterating the minimizing delta debugging algorithm,” in *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, ser. A-TEST 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 57–60.

[19] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, “ORBS: Language-independent program slicing,” in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*. ACM, 2014, pp. 109–120.

[20] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, “ORBS and the limits of static slicing,” in *Proc. 15th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2015, pp. 1–10.

[21] S. Yoo, D. Binkley, and R. Eastman, “Seeing is slicing: Observation based slicing of picture description languages,” in *Proc. 14th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2014, pp. 175–184.

[22] N. E. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo, “Generalized observational slicing for tree-represented modelling languages,” in *Proc. 11th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*. ACM, 2017, pp. 547–558.

[23] D. Binkley, N. Gold, S. Islam, J. Krinke, and S. Yoo, “Tree-oriented vs. line-oriented observation-based slicing,” in *Proc. 17th IEEE Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*. IEEE, 2017, pp. 21–30.

[24] ——, “A comparison of tree- and line-oriented observational slicing,” *Empirical Softw. Eng.*, vol. 24, no. 5, pp. 3077–3113, 2019.

[25] D. Stepanov, M. Akhin, and M. Belyaev, “Reduktor: How we stopped worrying about bugs in kotlin compiler,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 317–326.