On the Caching Schemes to Speed Up Program Reduction

YONGQIANG TIAN, University of Waterloo, Canada and The Hong Kong University of Science and Technology, China
XUEYAN ZHANG, University of Waterloo, Canada
YIWEN DONG, University of Waterloo, Canada
ZHENYANG XU, University of Waterloo, Canada
MENGXIAO ZHANG, University of Waterloo, Canada
YU JIANG, Tsinghua University, China
SHING-CHI CHEUNG, The Hong Kong University of Science and Technology, China
CHENGNIAN SUN*, University of Waterloo, Canada

Program reduction is a highly practical, widely demanded technique to help debug language tools, such as compilers, interpreters and debuggers. Given a program P that exhibits a property ψ, conceptually, program reduction iteratively applies various program transformations to generate a vast number of variants from P by deleting certain tokens, and returns the minimal variant preserving ψ as the result.

A program reduction process inevitably generates duplicate variants, and the number of them can be significant. Our study reveals that on average 61.8% and 24.3% of the generated variants in two representative program reducers HDD and Perses, respectively, are duplicates. Checking them against ψ is thus redundant and unnecessary, which wastes time and computation resources. Although it seems that simply caching the generated variants can avoid redundant property tests, such a trivial method is impractical in the real world due to the significant memory footprint. Therefore, a memory-efficient caching scheme for program reduction is in great demand.

This study is the first effort to conduct systematic, extensive analysis of memory-efficient caching schemes for program reduction. We first propose to use two well-known compression methods, ZIP and SHA, to compress the generated variants before they are stored in the cache. Furthermore, our keen understanding on the program reduction process motivates us to propose a novel, domain-specific, both memory and computation-efficient caching scheme, *Refreshable Compact Caching* (RCC). Our key insight is two-fold: ① by leveraging the correlation between variants and the original program P, we losslessly encode each variant into an *equivalent*, *compact*, *canonical* representation; ② periodically, stale cache entries, which will never be accessed, are timely removed to minimize the memory footprint over time.

*Corresponding author.

Authors' addresses: Yongqiang Tian, yongqiang.tian@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1 and The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China; Xueyan Zhang, xueyan.zhang@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1; Yiwen Dong, yiwen.dong@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1; Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1; Mengxiao Zhang, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1, m492zhan@uwaterloo.ca; Yu Jiang, Tsinghua University, Haidian District, Beijing, China, 100084, jiangyu198964@126.com; Shing-Chi Cheung, scc@cse.ust.hk, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, ON, Canada, N2L 3G1.

1049-331X/2023/1-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

Our extensive evaluation on 31 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 61.8% and 24.3% in HDD and Perses, respectively; correspondingly, the runtime performance is notably boosted by 22.8% and 18.2%. With regard to the memory efficiency, all three methods use less memory than the state-of-the-art string-based scheme STR. Specifically, ZIP and SHA cut down the memory footprint by more than 80% and 90% in both Perses and HDD compared to STR; moreover, the highly-scalable, domain-specific RCC dominates peer schemes, and outperforms the SHA by 96.4% and 91.74% in HDD and Perses, respectively.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: program reduction, delta debugging, debugging

ACM Reference Format:
Yongqiang Tian, Xueyan Zhang, Yiwen Dong, Zhenyang Xu, Mengxiao Zhang, Yu Jiang, Shing-Chi Cheung, and Chengnian Sun. 2023. On the Caching Schemes to Speed Up Program Reduction. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2023), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Given a program P and a property ψ that P exhibits (*e.g.*, P triggers a bug in an interpreter when the interpreter is executing P), *program reduction* aims to produce a smaller program P' that still exhibits ψ by removing tokens irrelevant to ψ from P [34, 43, 45, 52, 53].

Various program reduction techniques have been proposed and widely used in many applications, especially in the development of language tools, *e.g.*, compilers, interpreters, debuggers and static program analyzers [7, 34, 38, 40, 43, 45, 52, 53]. For example, both the GCC and LLVM communities have explicitly recommended that a bug-triggering test program should be minimized before it is reported in the bug tracking systems [12, 32]. This is because that a bug-triggering test program in C needs to have only thirty lines of code on average [42]; whereas in practice, such a test program collected from real-world programs or generated by automated compiler testing techniques [2, 27–29, 46, 49, 51] usually has at least several thousand lines of code. Without program reduction, it is a challenging task for developers to investigate the bug reports. Furthermore, as highlighted by a recent article in SIGPLAN [11], program reduction facilitates numerous other applications in software engineering and programming languages, such as optimization [39], fuzzing [36], program understanding and slicing [3].

Unfortunately, program reduction is computationally expensive and can even take days to finish reducing a program [26, 51]. Thus, it is beneficial for all potential users to improve the efficiency of program reduction. Conceptually, program reduction maintains a minimal program min, which satisfies ψ throughout the program reduction process, and initially min is P. Program reduction ① applies different program transformations to generate a vast number of variants from min by strategically deleting certain tokens, ② tests each variant on whether or not it still preserves ψ, and ③ sets the variant preserving ψ as min; this process is repeated until min cannot be further minimized, and min is returned as the final result. In the above process, the procedure of checking a variant against ψ is referred to as *a query to the property* in this paper, and queries usually account for a major portion of the overall time spent by the program reduction [17]. Some existing program reduction techniques also attempt to avoid generating uninteresting variants to improve the efficiency of program reduction [17, 19, 34, 43].

To understand the bottleneck of program reduction in terms of efficiency, we dive into the process and investigate the generated variants. We reveal that on average 61.8% and 24.3% of the generated variants in HDD [34] and Perses [43] are duplicates in our benchmark, because different program transformations may generate exactly the same variant by deleting different tokens. In other words, a significant amount of time is spent checking unnecessary duplicate variants against

ψ. If we cache such variants to avoid the redundancy, the program reduction efficiency is likely to be improved.

The state-of-the-art caching scheme for program reduction is string-based caching [20], *i.e.*, caching variants as strings or sequences of tokens, referred to as STR. However, our study reveals that such a trivial approach does not scale especially when P is large, due to its impractical memory consumption, which also concerns C-Reduce [37]. A large program P, unfortunately, is rather common in practice; in such cases, program reducers are likely to crash due to Out-of-Memory Error (OOM). An effective and efficient caching scheme for program reduction is desirable.

In this study, we take the first step to explore memory-efficient caching schemes for program reduction via compression. Specifically, we first leverage the following two readily available compression techniques to compress the source code of variants and cache the compressed source code instead of the original, uncompressed source code.

- Zip algorithm is a widely used, lossless data compression technique that reduces the size of large texts. Before being added to the cache, the string-based representation of each variant is compressed using the popular, general-purpose ZLIB compression library [10, 13]. This caching scheme is referred to as ZIP.
- Hashing is yet a popular lossy data compression technique that maps strings of various sizes to fixed-size values. A hash code is computed from the string-based representation and then added to the cache; specifically, SHA512 is used in this work [14, 24], due to its strong guarantee of collision resistance. We refer to this caching scheme as SHA.

However, from our comprehensive evaluations we find that the these two techniques still suffer from monotonic increase of memory consumption, and scalability issues among different program reducers. Therefore, after analyzing the characteristics of program reduction algorithms and the generated variants in depth, we gain the following two key insights.

(1) Every variant is derived from the minimal program min during program reduction by deleting some tokens. Therefore, the sequence p_v of tokens in each variant v is a *subsequence* of the sequence p_{min} of tokens in min.
(2) As a result, when a program becomes the the new min, any variant v in the cache that is not a subsequence of this min, can be safely removed from cache as v will no longer be accessed.

Based on these two insights, we propose a novel, domain-specific, memory and computation-efficient caching scheme, namely, Refreshable Compact Caching (RCC).

RCC. Based on the first insight, RCC computes a *compact encoding* for each variant to be added to the cache. This encoding is a set of slicing intervals in p_{min} that assembles p_v. Such a *lossless* compression algorithm significantly reduces the memory footprint. When required, the compact encoding can be rapidly uncompressed back to the original program variant. By leveraging the second insight, RCC periodically refreshes the cache to avoid memory leaks. Specifically, upon finding a new min during the program reduction process, RCC identifies and removes the cached variants that will never be accessed in the rest of the process. Cache refreshing further reduces the memory footprint by avoiding accumulating cache entries over time, and it thus improves scalability.

Conceptually, the domain-specific RCC is advantageous in practice compared to general caching schemes. Unlike ZIP, RCC compresses a variant into an array of integers without encoding each individual token. In contrast to SHA, RCC is an information-lossless compression algorithm, enabling the refreshing of cache to avoid the monotonic increase of cache size in program reduction.

We have implemented the proposed caching schemes on top of HDD and Perses, two representative program reduction algorithms that are being actively studied. Our extensive evaluation on 31 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 61.8% and 24.3% in HDD and Perses, respectively. The runtime performance is notably boosted by 22.8% and 18.2%. As for the memory efficiency, caching scheme ZIP and SHA cut down the memory footprint by more than 80% and 90% in both HDD and Perses compared to the baseline STR. Furthermore, the highly-scalable, domain-specific RCC dominates peer schemes, and it outperforms the second-best SHA by 96.4% and 91.74% in HDD and Perses, respectively. In most cases, RCC only takes tens of KB in Perses and less than one MB in HDD, which is at least one order of magnitude smaller than SHA.

Contributions. This paper makes the following contributions.

- We make the first effort to conduct systematic, extensive analysis of memory-efficient caching schemes for program reduction. We propose three caching schemes that are effective in improving the memory performance of caching in program reduction. These caching schemes are agnostic to most program reduction algorithms, and can be easily integrated into various program reduction tools and combined with other program reduction techniques, benefiting a great variety of researchers and developers.
- We propose a domain-specific caching scheme for program reduction. By leveraging the keen knowledge that variants are subsequences of the minimal program, RCC combines the compact encoding and cache-refresh algorithm to drastically reduce the memory footprint with great scalability. We formally prove the safety of refreshable cache and confirm with our evaluation.
- Our comprehensive evaluations on 31 real-world C compiler bugs demonstrate that caching help avoid issuing redundant queries by 61.8% and boost the runtime by 22.8%. Caching schemes ZIP and SHA cut down the memory footprint by 84.34% and 99.72% against the baseline; the domain-specific RCC further outperforms the second-best SHA by 91.74%.
- We have made our implementation, benchmarks, and evaluation scripts publicly available for reproducibility and replicability at https://github.com/uw-pluverse/perses/tree/master/doc/RCC.md

2 PRELIMINARIES

2.1 Sequences

This section introduces preliminary knowledge about sequences, since, in the rest of the paper, a program is represented as a sequence of tokens.

Let Σ be a set of elements. A *sequence* is an ordered list of elements denoted as $p = \langle t_1, t_2, \cdots, t_n \rangle$, where $t_i \in \Sigma \wedge (1 \leq i \leq n)$. Notation-wise,

`index`	$p[i]$ denotes t_i, the i-th element in p; i starts from 1.
`size`	$\|p\|$ denotes the number n of elements in p, which is also referred to as the *size* of p.
`slice`	$p[i:j]$ $(i \leq j \leq \|p\| + 1)$ represents a sequence $\langle t_i, t_{i+1}, \cdots, t_{j-1} \rangle$, a continuous slice of p starting from $p[i]$ inclusively and ending at $p[j]$ exclusively.
`concatenation`	given $p_1 = \langle t_1^1, t_2^1, \cdots, t_m^1 \rangle$ and $p_2 = \langle t_1^2, t_2^2, \cdots, t_n^2 \rangle$, $p_1 + p_2$ denotes the concatenation of p_1 and p_2, namely, $p_1 + p_2 = \langle t_1^1, t_2^1, \cdots, t_m^1, t_1^2, t_2^2, \cdots, t_n^2 \rangle$.
`equality`	$p_1 = p_2$ if $\|p_1\| = \|p_2\| \wedge \forall i \in [1, \|p_1\|]. p_1[i] = p_2[i]$

Definition 2.1 (Subsequence). *A sequence $p_1 = \langle t_1^1, t_2^1, \cdots, t_m^1 \rangle$ is a* subsequence *of another sequence $p_2 = \langle t_1^2, t_2^2, \cdots, t_n^2 \rangle$ if and only if there exists integers $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ where $t_1^1 = t_{i_1}^2$, $t_2^1 = t_{i_2}^2$, $\cdots$, $t_m^1 = t_{i_m}^2$. Notation-wise, this relation is written as $p_1 \sqsubseteq p_2$.*

Example. $\langle 1, 5 \rangle \sqsubseteq \langle 1, 3, 5 \rangle$, and $\langle 11, 3, 5 \rangle \sqsubseteq \langle 11, 3, 5 \rangle$.

Definition 2.2 (Proper Subsequence). *A sequence p_1 is a* proper subsequence *of another program p_2 if and only if $p_1 \sqsubseteq p_2 \wedge |p_1| < |p_2|$. Notation-wise, this relation is written as $p_1 \sqsubset p_2$.*

Example. $\langle 3 \rangle \sqsubset \langle 1, 3, 5 \rangle$, and $\langle 1, 5 \rangle \sqsubset \langle 1, 3, 5 \rangle$.

Lemma 2.1 (Transitivity). *Given three sequences p_1, p_2 and p_3, $p_1 \sqsubseteq p_2 \wedge p_2 \sqsubseteq p_3 \Rightarrow p_1 \sqsubseteq p_3$; similarly, $p_1 \sqsubset p_2 \wedge p_2 \sqsubset p_3 \Rightarrow p_1 \sqsubset p_3$.*

Example. Given that $\langle 1 \rangle \sqsubset \langle 1, 3 \rangle$, and $\langle 1, 3 \rangle \sqsubset \langle 0, 1, 2, 3 \rangle$, the transitivity of the proper subsequence implies $\langle 1 \rangle \sqsubset \langle 0, 1, 2, 3 \rangle$.

Definition 2.3 (Lexicographic Order). *Given two sequences of numbers p_1 and p_2, $p_1 < p_2$ if and only if p_1 and p_2 satisfy one of the following conditions.*

- $\exists i \in [1, \min(|p_1|, |p_2|)].\, p_1[1:i] = p_2[1:i] \wedge p_1[i] < p_2[i]$
- $|p_1| < |p_2| \wedge (p_1 = p_2[1 : |p_1| + 1])$

Example 1. $\langle 1, 2, 3 \rangle < \langle 1, 3, 3 \rangle$, since the first condition is satisfied. When $i = 2$, $p_1[1 : i = 2] = p_2[1 : i = 2] = \langle 1 \rangle$, and $p_1[i = 2] = 2 < p_2[i = 2] = 3$.

Example 2. $\langle 1, 2 \rangle < \langle 1, 2, 3 \rangle$, since the second condition is satisfied. Specifically, $|p_1| = 2 < |p_2| = 3$ and $p_1 = p_2[1 : 3] = \langle 1, 2 \rangle$.

2.2 Program Reduction

In this paper, a program is represented as a sequence of tokens, $\langle t_1, t_2, \cdots, t_n \rangle$, where t_i $(1 \leq i \leq n)$ is a token. Given a program P with a property of interest, $\mathbb{P}$ denotes the search space of all the possible variants derivable from P by deleting some tokens, that is, $\forall p \in \mathbb{P} : p \sqsubseteq P$. Let $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ and $p \in \mathbb{P}$, then the property can be defined as a function $\psi(p) : \mathbb{P} \rightarrow \mathbb{B}$, where

$$\psi(p) = \begin{cases} \mathsf{true} & \text{if } p \text{ exhibits the property} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

2.2.1 Deletion-Based Program Transformation. We use $\mathbb{T}$ to denote a set of deletion-based program transformations. Formally, a deletion-based program transformation $\tau \in \mathbb{T}$ is defined as a function $\tau : \mathbb{P} \rightarrow \mathbb{P}$, which generates a new program by removing tokens from the non-empty input program. Mathematically, $|p| > 0 \wedge p \in \mathsf{dom}(\tau) \Rightarrow \tau(p) \sqsubset p$, where the domain of τ, $\mathsf{dom}(\tau)$, represents the universe of candidate programs that can be accepted as inputs of the transformation τ and $p \in \mathsf{dom}(\tau)$ implies that the program transformation τ is applicable on the program p.

In this paper, we focus on deletion-based program transformations, because most state-of-the-art program reduction algorithms only support this category of program transformations, such as Delta Debugging (DD) [52], Hierarchical Delta Debugging (HDD) [34], Generalized Tree Reduction (GTR) [17], Chisel [16], and Perses [43].

One exception is C-Reduce which supports program transformations out of this category [38]. For example, C-Reduce uses Clang [33] to inline function calls to reduce the number of function definitions, which increases the size of variants. However, the number of such program transformations is small, and the main program transformations supported in C-Reduce are still deletion-based, *e.g.*, DD and HDD.

2.2.2 Program Reduction without Cache. Algorithm 1 lists the common, overall workflow of program reduction. Most language-agnostic program reduction algorithms [16, 17, 34, 40, 43, 52] follow this workflow, as long as these algorithms transform programs by deleting tokens. For example, DD, HDD, Perses, and Chisel generate variants by deleting tokens, and all their concrete workflows can be conceptually generalized to Algorithm 1, though the differences in determining

Algorithm 1: Conceptual Workflow of Program Reduction

```
Input: $P$: the program to be reduced.
Input: $\psi : \mathbb{P} \to \mathbb{B}$: the property of interest.
Output: A minimal program $\mathsf{min} \in \mathbb{P}$ s.t. $\psi(\mathsf{min})$
1 $\mathbb{T}$: a set of deletion-based program transformations defined in §2.2.1
2 $\mathsf{min} \leftarrow P$
3 while true do
4     $\mathsf{prev} \leftarrow \mathsf{min}$
5     for $\tau \in \mathbb{T}$ do
6         if $\mathsf{min} \notin \mathrm{dom}(\tau)$ then continue
7         $p \leftarrow \tau(\mathsf{min})$
8         if $\psi(p)$ then $\mathsf{min} \leftarrow p$
9     if $|\mathsf{prev}| = |\mathsf{min}|$ then return $\mathsf{min}$
```

what tokens to delete are significantly divergent between the aforementioned program reduction algorithms. $\mathbb{T}$ on line 1 denotes an abstract set of deletion-based program transformations described in §2.2.1. The concrete program transformations in $\mathbb{T}$ depend on the concrete program reducer; *e.g.*, Perses supports more types of deletion-based program transformations than DD and HDD.

Note that the workflow in Algorithm 1 is widely applicable, even to C-Reduce if we relax $\mathbb{T}$ to include non-deletion-based program transformations supported by C-Reduce.

2.2.3 Program Reduction with String-Based Cache. Algorithm 2 presents a general workflow of program reduction with a string-based cache (referred to as STR) enabled [20], where each variant is represented by its source code. The major differences from Algorithm 1 are ① the introduction of the variable cache on line 3, ② the presence test of p in cache on line 10, and ③ adding the program that fails the property test to cache on line 12.

The major drawback with Algorithm 2 is the vast memory footprint induced by cache because each program in cache is represented with its source code (*viz.*, line 9), and cache is monotonically growing due to line 12. This problem can be exacerbated when the program to be reduced is large. For example, to reduce subject clang-27137 with 174,538 tokens in Table 5, STR requires 698.91 MB memory to cache variant programs in Perses; due to differences in supported program transformations, it incurs a much larger memory footprint in HDD, *i.e.*, 98.67 GB. Such a large memory overhead on one subject is impractical in the production environment, prohibiting deploying multiple instances of program reduction in a single machine.

3 A MOTIVATING EXAMPLE

We illustrate how duplicate programs are generated during the program reduction process with a contrived example in Figure 1. This example includes one original program in Figure 1a, and a property of interest that the program exits with zero returned. Figure 1b–1j show nine variants sequentially generated in the program reduction process; note that the program reduction process is simplified for illustrative purposes from a real program reduction process by Perses by ignoring the other less interesting generated variants.

Step 0: Initially, the minimal program min is the original input p_0 in Figure 1a, and this program exits with zero.

Step 1–3: Three variants as shown in Figure 1b, Figure 1c and Figure 1d are generated from min by removing one or more statements, but none of them is semantically valid *w.r.t.* the C language specification and thus not of interest. Note that the program p_3 in Figure 1d is generated for the first time, and will be repeatedly generated later.

Algorithm 2: Program Reduction with STR

```
Input: $P$: the program to be reduced.
Input: $\psi : \mathbb{P} \rightarrow \mathbb{B}$: the property of interest.
Output: A minimal program min $\in \mathbb{P}$ s.t. $\psi$(min)
$\mathbb{T}$: a set of deletion-based program transformations defined in §2.2.1
min ← $P$
cache ← $\varnothing$
while true do
    prev ← min
    for $\tau \in \mathbb{T}$ do
        if min $\notin$ dom($\tau$) then continue
        $p \leftarrow \tau$(min)
        cache_key ← a string which is the source code of $p$
        if cache_key ∈ cache then continue // $p$ has been tested before.
        if $\psi(p)$ then min ← $p$
        else cache = cache ∪ {cache_key} // $p$ does not preserve $\psi$, and is thus cached.
    if |prev| = |min| then return min
```

```
int main() {
    int a = 0;
    int b = 9;
    return a + 0;
}
```

(a) $p_0, \psi(p_0)$ = true

```
int main() {



}
```

(b) $p_1, \psi(p_1)$ = false

```
int main() {

    int b = 9;
    return a + 0;
}
```

(c) $p_2, \psi(p_2)$ = false

```
int main() {
    int a = 0;


}
```

(d) $p_3, \psi(p_3)$ = false

```
int main() {
    int a = 0;

    return a + 0;
}
```

(e) $p_4, \psi(p_4)$ = true

```
int main() {
    int a = 0;


}
```

(f) $p_5, \psi(p_5)$ = false

```
int main() {


    return a + 0;
}
```

(g) $p_6, \psi(p_6)$ = false

```
int main() {
    int a = 0;

    return 0;
}
```

(h) $p_7, \psi(p_7)$ = true

```
int main() {
    int a = 0;


}
```

(i) $p_8, \psi(p_8)$ = false

```
int main() {


    return 0;
}
```

(j) $p_9, \psi(p_9)$ = true

Fig. 1. A contrived, illustrative example of a program reduction process. Figure (a) shows the original program, and the property of interest is that the program returns zero. Figures (b)–(j) are nine variants sequentially generated during the program reduction process. Figures (d), (f), and (i) with captions in blue show duplicate variants, and Figures (a), (e), (h) and (j) with captions in lime show the minimal variants satisfying the property.

Step 4: Another variant p_4 is derived from min, and satisfies the property, and thus p_4 becomes the new min. Any new variant in the future will be generated from p_4.
Step 5, 6: Two variants p_5 and p_6 are generated from p_4 by deleting one statement, but neither of them satisfies the property. However, p_5 is duplicate to p_3, and this duplicate incurs an unnecessary query to the property.
Step 7: The variant p_7 in Figure 1h is generated from p_4 by deleting two tokens a and + from the return statement `return a + 0;`. This variant preserves the property and becomes the new min.

Step 8: From the new minimal program p_7, p_8 as shown in Figure 1i is generated by deleting the return statement, and this is the third time the same variant is generated. Without caching, this variant issues another redundant query to the property.
Step 9: The variant p_9 is generated by deleting the variable definition from p_7, and this is the final result of the program reduction.

In this contrived program reduction run, a program as shown in Figure 1d is generated three times, and it requires three queries to the property of which two are redundant. As mentioned in §1, queries to the property account for the majority of the program reduction time. It will be desirable to eliminate such redundant queries to shorten the program reduction time with memory efficient caching scheme, the focus of this paper.

3.1 Caching Program Variants in Program Reduction

This section briefly describes how caching helps avoid redundant property queries, and how ZIP, SHA, and RCC reduces memory footprint compared to Algorithm 2 proposed in literature [20].

STR. Algorithm 2 prevents redundant queries by saving the source code of the variants that do not satisfy the property in cache. For example, p_3 in Figure 1d is represented as the following string by the encoding Algorithm 2.

“int␣main␣(␣)␣{␣int␣b␣=␣9␣;␣return␣a␣+␣0␣;␣}”

In Java, this string object takes up at least 86 bytes excluding the meta data added by the Java Virtual Machine, *i.e.*, 86 bytes from the 43 characters (a character in Java is two bytes).

ZIP. To reduce the memory footprint of the trivial string representation, we exploit the popular ZLIB library [10, 13], a lossless compression algorithm. It effectively compresses the string representation into a byte array. For example, ZIP compresses p_3 to a byte array of 48 elements.

SHA. We investigate another popular, more aggressive but lossy compression technique, hash algorithm. Specifically, the hash function SHA-512 produce a 512-bit digest from the string representation [14, 24], *e.g.*, SHA hashes p_3 into a 512-bit digest (64 bytes) in Java.

RCC. By leveraging keen insights in program reduction, we propose a domain-specific caching scheme RCC to efficiently avoid redundant property queries. In RCC, p_3 is ever encoded as a compact representation, $\langle\!\langle 1, 11, 21, 22\rangle\!\rangle$, $\langle\!\langle 1, 11, 16, 17\rangle\!\rangle$ or $\langle\!\langle 1, 11, 14, 15\rangle\!\rangle$ throughout the program reduction process.[1]

The details of the encoding process will be introduced in §4.3.2. Intuitively, every two integers in the array correspond to a continuous range of tokens in min. For example, in $\langle\!\langle 1, 11, 21, 22\rangle\!\rangle$, 1 and 11 refer to min$[1 : 11]$; 21 and 22 refers to min$[21 : 22]$. At any time during program reduction, the cache key of p_3 only occupies 16 bytes ($4 * 4$, each int in Java is 4-byte), compared to the 86 bytes by STR.

The other key feature of RCC is *refreshable caching*. RCC is able to determine whether a variant will never be generated in the future. If yes, such a variant will be removed from cache. For example, at the time when p_4 in Figure 1e is being generated, cache = $\{p_1, p_2, p_3\}$; after p_4 is tested to satisfy the property and set as min, RCC is able to accurately predict that p_2 will never be generated, and thus removes p_2 from cache, which makes cache = $\{p_1, p_3\}$.

[1]Such a compact representation, named as “interval-based encoding”, is introduced in details in §4.3.2. An interval-based encoding is a sequence of integers. For readability purpose, we use $\langle\!\langle t_1, t_2, \cdots, t_n\rangle\!\rangle$ to represent an interval-based encoding and use $\langle t_1, t_2, \cdots, t_n\rangle$ to represent a sequence.

3.2 Challenges of Caching Variants during Program Reduction

In practice, caching variants is usually complicated and challenging. Unfortunately, large programs as input to program reduction are rather common, as these programs are either collected from real-world software or generated by automated testing techniques [2, 27, 29, 46, 49, 51].

When the initial program P is large, the total number of queries usually increases significantly, and thus the difference in memory footprint between different caching schemes can be amplified. For example, the subject clang-27137 in Table 5 has 173,538 tokens, HDD issues as much as 720,875 queries. HDD with STR exhausts a memory heap of 44 GB, and eventually crashes with OOM. With ZIP, HDD successfully finishes the program reduction process, consuming 18.7 GB of memory. Given the consistent digest size, SHA is sensitive to the number of queries and requires 85.8 MB to reduce the subject. Exceedingly, RCC demands only 4.4 MB at peak.

4 METHODOLOGIES

This section details the design of the three caching schemes proposed in this paper, namely ZIP, SHA, and RCC. The main objective is to reduce the memory footprint of the program representation without noticeable runtime overhead, such that each cache key is compact in size within the cache. To the best knowledge of the authors, this is the first effort to conduct systematic, extensive analysis of memory-efficient caching schemes to speed up program reduction.

4.1 Lossless Compression: ZIP

Zip algorithm is a lossless compression technique representative, which effectively compresses data. ZLIB is a well-known, general-purpose lossless data compression library, which is widely used across different platforms (*e.g.*, Linux, macOS, and iOS) [10, 13]. The main algorithm, DEFLATE, is capable of compressing a variety of data with limited system resources. Additionally, there is no theoretical limitation to the data size being compressed.

ZIP cache scheme compresses the string representation of a variant program into a byte array, which is then used as the cache key. Note that ZLIB provides controls to computing resources, and we prefer better compression level for minimal memory footprint rather than the speed of compression. §5.3 shows that the runtime overhead of the way we use ZLIB is practically negligible.

4.2 Lossy Compression: SHA

Hash algorithm is an irreversible process of converting data into hash values of fixed length (*a.k.a.*, digest). The original data cannot be recovered; thus, hash algorithm is a lossy compression technique. Hash algorithms are widely used in internet security and digital certificates, but we are interested in applying it to the string representation of a variant program.

We adopted SHA-512 over alternative hash functions for two main reasons. ① Secure hash algorithm (SHA) is well supported by available libraries and easy to deploy. ② SHA512 provides the strongest guarantee of collision resilience, where different string inputs are less likely to have the same digest [14, 24]. Note that even the collision chance is slim, if hash collision ever occurs, it is possible for a program reducer to produce a different program reduction result, which could be sub-optimal. SHA consistently compresses the string representation of variant programs of different sizes into a 512-bit digest (64 bytes), which is then used as the cache key.

Table 1. Examples of Encoding

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Token	`int`	`main`	`(`	`)`	`{`	`int`	`a`	`=`	`0`	`;`	`int`	`b`	`=`	`9`	`;`	`return`	`a`	`+`	`0`	`;`	`}`
p_1	•	•	•	•	•																•
	Encoding = $\langle\langle 1, 6, 21, 22 \rangle\rangle$																				
p_2	•	•	•	•	•	•						•	•	•	•	•	•	•	•	•	•
	Encoding = $\langle\langle 1, 7, 12, 22 \rangle\rangle$																				
p_3	•	•	•	•	•	•	•	•	•	•											•
	Encoding = $\langle\langle 1, 11, 21, 22 \rangle\rangle$																				
p_4	•	•	•	•	•	•	•	•	•	•						•	•	•	•	•	•
	Encoding = $\langle\langle 1, 11, 16, 22 \rangle\rangle$																				

(a) Encoding *w.r.t.* p_0

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Token	`int`	`main`	`(`	`)`	`{`	`int`	`a`	`=`	`0`	`;`	`return`	`a`	`+`	`0`	`;`	`}`
p_1	•	•	•	•	•											•
	Encoding = $\langle\langle 1, 6, 16, 17 \rangle\rangle$															
p_3 p_5	•	•	•	•	•	•	•	•	•	•						•
	Encoding = $\langle\langle 1, 11, 16, 17 \rangle\rangle$															
p_6	•	•	•	•	•						•	•	•	•	•	•
	Encoding = $\langle\langle 1, 6, 11, 17 \rangle\rangle$															
p_7	•	•	•	•	•	•	•	•	•	•	•			•	•	•
	Encoding = $\langle\langle 1, 12, 14, 17 \rangle\rangle$															

(b) Encoding *w.r.t.* p_4

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Token	`int`	`main`	`(`	`)`	`{`	`int`	`a`	`=`	`0`	`;`	`return`	`0`	`;`	`}`
p_1	•	•	•	•	•									•
	Encoding = $\langle\langle 1, 6, 14, 15 \rangle\rangle$													
p_3 p_5 p_8	•	•	•	•	•	•	•	•	•	•				•
	Encoding = $\langle\langle 1, 11, 14, 15 \rangle\rangle$													
p_9	•	•	•	•	•						•	•	•	•
	Encoding = $\langle\langle 1, 6, 11, 15 \rangle\rangle$													

(c) Encoding *w.r.t.* p_7

These three tables show the encoding process with respect to base program p_0, p_4, and p_7 respectively. The first row is the indices starting from one, and the second row lists the corresponding tokens of the min programs. A continuous region of bullets with colored background shows the interval of the compact interval-based encoding. For instance, p_2 in (a) indicates that a variant, p_2, is derived from p_0 by deleting successive nodes from 7 to 11, and the consecutive regions, marked with bullets, are encoded with the starting and ending node indices. Therefore, the encoding of p_2 *w.r.t.* p_0 is $\langle\langle 1, 7, 12, 22 \rangle\rangle$.

4.3 Domain-Specific Compression: RCC

Finally, this section describes the design and algorithms of RCC, a novel, domain-specific, memory-efficient caching scheme for program reduction. RCC includes two key concepts *compact encoding* and *refreshable caching*, both of which are based on the following insights neglected in the literature.[2]

Insight 1 At any time during program reduction, let min be the minimal program found at that time (initially, min is P), then any variant p that is generated later is a subsequence of min, *i.e.*, $p \sqsubset \text{min}$.

Insight 2 For any program p, *s.t.*, $p \not\sqsubseteq \text{min}$, p will never be generated later by any deletion-based program transformation.

4.3.1 Overall Workflow with RCC.

Algorithm 3 lists the general workflow of program reduction with RCC. Compared to Algorithm 2, there are two major differences:

Compact Encoding as Cache Key. On line 9 Algorithm 3 calls `CompactEncode` to convert a program p to a *compact* (memory-efficient), *equivalent* representation as the cache key. `CompactEncode` takes as input not only p, but also min to compute this cache key, whereas the vanilla program reduction with string-based cache in Algorithm 2 uses the source code of p (a sequence of characters) as the cache key on line 9. The compact encoding scheme of RCC uses much less memory than STR, which will be detailed in §4.3.2.

Refreshable Caching. Algorithm 3 refreshes cache on line 13 when a new minimal program is found. The cache-refresh algorithm identifies programs that will not be generated afterward based on Theorem 4.2 and removes them from cache to reduce memory footprint. In contrast, the size of STR in Algorithm 2 monotonically increases, and therefore STR usually consumes a large amount of memory.

4.3.2 Compact Encoding of Programs.

[2]These two insights are equivalent with the insights in §1, but are re-illustrated using the annotation defined by us.

Algorithm 3: Program Reduction with RCC

```
Input: $P$: the program to be reduced.
Input: $\psi : \mathbb{P} \to \mathbb{B}$: the property of interest.
Output: A minimal program $\mathsf{min} \in \mathbb{P}$ s.t. $\psi(\mathsf{min})$
1  $\mathbb{T}$: a set of deletion-based program transformations defined in §2.2.1
2  $\mathsf{min} \leftarrow P$
3  $\mathsf{cache} \leftarrow \emptyset$
4  while true do
5      $\mathsf{prev} \leftarrow \mathsf{min}$
6      for $\tau \in \mathbb{T}$ do
7          if $\mathsf{min} \notin \mathrm{dom}(\tau)$ then continue
8          $p \leftarrow \tau(\mathsf{min})$
9          $cache\_key \leftarrow$ CompactEncode($\mathsf{min}, p$)
10         if $cache\_key \in \mathsf{cache}$ then continue
11         if $\psi(p)$ then
12             $\mathsf{min} \leftarrow p$
               // Refresh cache with the new minimal program.
13             $\mathsf{cache} \leftarrow$ RefreshCache($\mathsf{cache}, \mathsf{prev}, \mathsf{min}$)
14         else $\mathsf{cache} \leftarrow \mathsf{cache} \cup \{cache\_key\}$
15     if $|\mathsf{prev}| = |\mathsf{min}|$ then return $\mathsf{min}$
16 Function RefreshCache($old\_cache$, $\mathsf{prev}$, $\mathsf{min}$):
       Input: $old\_cache$: the cache used previously
       Input: $\mathsf{prev}$: the previous min
       Input: $\mathsf{min}$: the current/new min
17     $\mathsf{cache} \leftarrow \emptyset$
18     for $encoding \in old\_cache$ do
19         $p' \leftarrow$ CompactDecode($\mathsf{prev}, encoding$)
20         if $p' \not\sqsubseteq \mathsf{min}$ then continue
21         $\mathsf{cache} \leftarrow \mathsf{cache} \cup \{$CompactEncode($\mathsf{min}, p'$)$\}$
22     return cache
```

Definition 4.1 (Interval-Based Encoding). *Given the minimal program* min *as the base program and a program p,* s.t., $p \sqsubseteq \mathsf{min}$, *a sequence e of integers is an interval-based encoding of p* w.r.t. *the base program* min, *if and only if e satisfies all the following properties,*

(1) e has an even number of elements
(2) $\forall i \in [1, |e|).\, e[i] < e[i+1]$
(3) $\sum_{i=1}^{|e|/2} \mathsf{min}[e[2i-1] : e[2i]] = \mathsf{min}[e[1] : e[2]] + \cdots + \mathsf{min}[e[|e|-1] : e[|e|]] = p$

$\sum_{i=1}^{n} s_i = s_1 + s_2 + \cdots + s_n$ represents a sequence by concatenating $s_1, s_2, \cdots,$ and s_n. Please note that since the interval-based encoding of p is a sequence of integers, all the definitions and operations of sequences mentioned in §2.1 are also applicable to interval-based encoding, including `index`, `size`, `slice` and so on. For readability purposes, we use $\langle\!\langle t_1, t_2, \cdots, t_n \rangle\!\rangle$ to represent an interval-based encoding.

Example. As shown in Table 1a, the interval-based encoding of p_1 *w.r.t.* p_0 is $\langle\!\langle 1, 7, 12, 22 \rangle\!\rangle$.

Definition 4.2 (Padding). *Given an interval $[a, b)$ where $a < b$, the function $pad(a, b)$ denotes a continuous sequence $p = \langle a, a+1, \cdots, b-1 \rangle$ by padding the interval with the missing numbers. Formally, $a = p[1]$, $b - 1 = p[|p|]$, and $\forall i \in [1, |p|-1].p[i] + 1 = p[i+1]$.*

Example. Given an interval $[1, 4)$, $pad(1, 4) = \langle 1, 2, 3 \rangle$.

Definition 4.3 (Encoding Expansion). *Given an interval-based encoding e and $i \in [1, |e|/2]$, the encoding expansion operator $expand()$ applies $pad()$ to every interval $[e[2i-1], e[2i])$, namely,*

$$expand(e) = \sum_{i=1}^{|e|/2} pad(e[2i-1], e[2i]) = pad(e[1], e[2]) + \cdots + pad(e[|e|-1], e[|e|])$$

Example. Assuming a program has the interval-based encoding $e = \langle\!\langle 1, 4, 6, 9 \rangle\!\rangle$, then $expand(e) = pad(1, 4) + pad(6, 9) = \langle 1, 2, 3 \rangle + \langle 6, 7, 8 \rangle = \langle 1, 2, 3, 6, 7, 8 \rangle$.

Definition 4.4 (Canonical Encoding). *Given the minimal program* min*, a program p, and an interval-based encoding e of p* w.r.t. min, *e is canonical if and only if $expand(e)$ is lexicographically minimum among all interval-based encodings of p* w.r.t. min, i.e., *$\nexists\, e'.\, expand(e') < expand(e)$. Note that $expand(e') < expand(e)$ is the lexicographic order defined in definition 2.3.*

Example. Table 1 lists three sets of encodings *w.r.t.* three different base programs, and Table 1a shows the compact encoding of four programs *w.r.t.* p_0. We take p_2 as a concrete example to illustrate definition 4.1 and definition 4.4. In Table 1a, the canonical interval-based encoding of p_2 is a compact array $e = \langle\!\langle 1, 7, 12, 22 \rangle\!\rangle$:

(1) e has an even number of elements (*i.e.*, $|e| = 4$).
(2) the elements in e are sorted in ascending order.
(3) the concatenation of $p_0[e[1] : e[2]]$ and $p_0[e[3] : e[4]]$ equals p_2, that is, $p_0[e[1] : e[2]] + p_0[e[3] : e[4]] = p_0[1 : 7] + p_0[12 : 22] = p_2$.
(4) e is the canonical encoding by definition 4.4. There is no other interval-based encoding e' such that $expand(e')$ lexicographically less than $expand(e)$.
Note that p_2 is generated from p_0 by deleting `int a = 0;` (*i.e.*, deleting $p_0[6 : 11]$ from p_0), and we can obtain the following origin information: $p_2[1 : 6]$ from $p_0[1 : 6]$, and $p_2[6 : 17]$ from $p_0[11 : 22]$. This origin information can also be encoded as a compact array $e'' = \langle\!\langle 1, 6, 11, 22 \rangle\!\rangle$, which satisfies the interval-based encoding in definition 4.1. However, e'' is not the canonical encoding for p_2 because of $expand(e) < expand(e'')$.

4.3.3 Evolution of Encoding. We refer to the canonical interval-based encoding as **compact encoding** in the rest of the paper. Specifically, the compact encoding of a program p is computed over a base program min. For different base programs, the same program can have different encodings. For example, in Table 1a the encoding of p_3 *w.r.t.* p_0 is $\langle\!\langle 1, 11, 21, 22 \rangle\!\rangle$, whereas in Table 1b its encoding *w.r.t.* p_4 is $\langle\!\langle 1, 11, 16, 17 \rangle\!\rangle$ and the one *w.r.t.* p_7 is $\langle\!\langle 1, 11, 14, 15 \rangle\!\rangle$ in Table 1c.

The function `CompactEncode` in Algorithm 4 computes the compact encoding of p *w.r.t.* min. Starting from line 5, it iterates through p from the head. For each element $p[i]$, `CompactEncode` locates the first element matching $p[i]$ in min from the position `min_index` on line 6~line 8. Please note that `CompactEncode` has found the start of an interval (*i.e.*, `min_index` on line 8). In the following line 9~line 11, this function searches for the end of the current interval by continuously advancing both i and `min_index`, until `min_index` has reached the end of min or `min[min_index]`$\neq p[i]$ on line 9; when the loop exits, `min_index` is the end of the current interval, and added to the compact encoding on line 12. Note that the parameter p of `CompactEncode` is a proper subsequence of min, so `CompactEncode` always returns a valid canonical interval-based encoding.

The function `CompactDecode` is straightforward, as it reconstructs the program p from its compact encoding by interpreting definition 4.1, especially the third condition in the definition, *i.e.*, $\sum_{i=1}^{|e|/2} \text{min}[e[2i-1] : e[2i]] = p$.

Time Complexity. Both algorithms are linear in terms of time complexity. In particular, the time complexity of `CompactEncode` is $O(|p| + |\text{min}|)$, and `CompactDecode` is $O(|encoding| + |\text{min}|)$.

Algorithm 4: Compact Encoding and Decoding

```
Function CompactEncode(min, $p$):
    Input: min: a program, s.t., $p \sqsubset$ min
    Input: $p$: a program to be encoded
    Output: The canonical, compact encoding of $p$ w.r.t. min
    result ← []
    min_index ← 1
    $i \leftarrow 1$
    while $i <= |p|$ do
        // scan for the start of the next interval.
        while min[min_index] $\neq p[i]$ do
            min_index ← min_index + 1
        result ← result + [min_index]
        // scan for the exclusive end of the next interval.
        while min_index $\leq$ |min| $\wedge$ min[min_index] = $p[i]$ do
            min_index ← min_index + 1
            $i \leftarrow i + 1$
        result ← result + [min_index]
    return result
Function CompactDecode(min, $encoding$):
    Input: min: a program
    Input: $encoding$: a canonical, compact encoding of a program $p$ w.r.t. min
    Output: $p$: the program of which $encoding$ is w.r.t. min
    $p \leftarrow []$
    for $i \leftarrow 1$ to $|encoding|/2$ do
        $start \leftarrow encoding[2 * i - 1]$
        $end \leftarrow encoding[2 * i]$
        $p \leftarrow p$ + min[$start : end$]
    return $p$
```

4.3.4 Cache Refresh. Throughout the whole duration of program reduction, there is a continuously updated minimal program min which satisfies ψ. Initially min is P; the size of min is monotonically decreasing, because all variants are generated from min (*viz.*, line 8 in Algorithm 3) and min is updated to the variant satisfying ψ on line 12 in Algorithm 3; in the end min is the final result of program reduction. Based on the procedure above, we have the following property of min.

Lemma 4.1 (Subsequence Relation of Minimal Programs). *Let* min_i *denote the minimal program at time* t_i*, and* min_j *denote the minimal one at time* t_j*, where* $\text{min}_i \neq \text{min}_j$ *and* $t_i < t_j$*. Then* min_j *is a proper subsequence of* min_i, i.e., $\text{min}_j \sqsubset \text{min}_i$.

Proof. In Algorithm 3, each minimal program is derived from its previous minimal program (*viz.*, line 8 and line 12). Therefore, the history of values of min from min_i to min_j can be represented as a sequence $h = \langle \text{min}_i, \text{min}_{i+1}, \text{min}_{i+2}, \cdots, \text{min}_j \rangle$, where $\forall k \in [1, |h|) : h[k+1] \sqsubset h[k]$. Based on the transitivity property in lemma 2.1, we can prove $\text{min}_j \sqsubset \text{min}_i$. □

Example. in the contrived program reduction process in Figure 1, min has four values from the start of the program reduction till the end, *i.e.*, p_0, p_4, p_7 and p_9. It is trivial to see $p_9 \sqsubset p_7 \sqsubset p_4 \sqsubset p_0$.

Theorem 4.2 (Safety of Cache Refresh). *Let* min *denote the minimal program at any time* t *during a program reduction process. If* $p \not\sqsubset$ min*, then* p *will never be generated by any program transformation in* $\mathbb{T}$ *in the remainder of the program reduction process after* t.

Proof. Proof by contradiction. Assume p can be generated by $\tau \in \mathbb{T}$ from the minimal program min' at time t' ($t' > t$), *i.e.*, $p = \tau(\mathsf{min}')$. As τ is a program transformation which deletes tokens from min', we have $p \sqsubset \mathsf{min}'$. Based on lemma 4.1, we know $\mathsf{min}' \sqsubset \mathsf{min}$. Based on the transitivity of the subsequence relation in lemma 2.1, we can further conclude $p \sqsubset \mathsf{min}$, which contradicts the condition $p \not\sqsubset \mathsf{min}$ in the theorem. □

Again in Figure 1, right after p_2 is generated and tested not to satisfy the property, p_2 is added to cache. But when the second min variant p_4 is found, we see that p_2 is not a subsequence of p_4, and according to Theorem 4.2, we can safely remove p_2 from cache. Moreover, we cannot compute a compact encoding for p_2 *w.r.t.* either p_4 or p_7. This is also why in Table 1, p_2 only appears in Table 1a *w.r.t.* p_0, but not in the other two tables. Notice that the lossless compression of RCC is the necessary foundation of the cache-refresh algorithm. Lossy compression such as SHA cannot adopt such a cache-refreshing algorithm because it is impossible to determine if a variant is a proper subsequence of other variant a by comparing their hash digests.

5 EVALUATION

We conducted comprehensive evaluations to demonstrate the advantages of the proposed caching schemes in the following aspects: ① memory efficiency, ② effectiveness in speeding up program reduction, and ③ generality to work with different program reduction algorithms. We also conducted ablation experiments to investigate the effect of the two main components of RCC: compact encoding and cache refreshing. Additionally, we investigated two different cache refreshing mechanisms.

5.1 Experiment Design

5.1.1 Baseline. Our baseline is the string-based caching (STR) algorithm discussed in §2.2.3, the state of the art proposed in literature [20]. We implemented the proposed caching schemes (ZIP, SHA, and RCC) on top of Perses [43], the state-of-the-art language-agnostic program reduction algorithm. To demonstrate the generalizability of RCC, we also implemented RCC on HDD [34], one of the representative, legendary algorithms in program reduction. The core idea of HDD is adopted by many subsequent studies [18, 19, 25, 38], including Perses. Compared to HDD, Perses leverage grammar to avoid generating syntactically invalid program in reduction.

5.1.2 Research Questions. We aim to answer the following research questions in our evaluation.
RQ1 (Memory Efficiency): *Which caching scheme demonstrates the best memory efficiency in program reduction?*

We measure and compare the memory footprint of STR, ZIP, SHA, and RCC in HDD and Perses. Concretely, for each caching scheme, we measure the peak cache size and the average cache size. The former is the highest amount of memory consumption in the reduction, while the latter provides a general view of memory consumption over time.
RQ2 (Program Reduction Efficiency): *To what extent can caching schemes improve the efficiency of program reduction?*

We run HDD and Perses without caching or with different caching schemes on the benchmark and measure their wall time and the number of queries. We then particularly compare the program reduction runs without caching to those with caching in terms of program reduction efficiency (*i.e.*, number of queries and reduction time). Moreover, by comparing the number of queries using STR and RCC, we can also validate the safety of cache refreshing in RCC, *i.e.*, none of the removed entries in RCC will be generated in the subsequent program reduction process.

RQ3 (Effect of Compact Encoding and Refreshable Caching): *What are the effects of compact encoding and refreshable caching of RCC in program reduction?*

To study compact caching, the lossless, domain specific compression technique in RCC, we implement CC scheme by disabling cache refreshing in RCC on top of Perses. We then compare Perses+CC against other schemes to contrast the effect of compact encoding. We implemented Perses+RSTR by adding cache refreshing to STR, and then plot the memory consumption over time to observe the memory footprint changes before and after cache refreshing events.

5.1.3 Evaluation Settings.

Benchmark Suite. To reassemble a realistic workload for program reduction algorithms, we build a C program benchmark consisting of 31 subjects. 20 of them are collected by Perses [43] from the official bug repositories of GCC and Clang, and they are later used in literature [16, 48, 50]. We further collect 11 from the official bug repositories of GCC and Clang. These 31 programs have 79,941.7 tokens on average, and each of them triggers either a crash or miscompilation bug in at least one stable compiler release. To the best knowledge of the authors, this is the largest benchmark suite from real C compiler bug repositories. The sizes of these subjects and the diverse types of bugs make this benchmark simulate a representative application scenarios of program reductions in the real world. Table 2 shows the detailed information of each subject in our benchmark suite.

Experiment Environment. To simulate the computation environment of software developers, all experiments were conducted on cloud virtual machines running Ubuntu 20.04 LTS. Each virtual machine is configured with AMD EPYC 7763v CPU and 384 GB RAM.

Cache Profiling. We used `ObjectExplorer` [1] to measure the memory footprint of the cache object. Since memory profiling is time-consuming and can introduce runtime overhead, we conducted evaluations of memory footprint and time measurement in separate runs.

5.2 RQ1: Memory Efficiency

We study memory efficiency in two aspects. First, we measure the peak memory footprint of different caching schemes to investigate their worst-case space complexity. Second, we measure the average cache size to understand the overall memory consumption over time.

5.2.1 Peak Cache Size. We first measured the peak cache size of each caching scheme. Table 3 shows the peak cache size in Kilobytes of different caching schemes in HDD and Perses.

STR. In both HDD and Perses, STR has the largest memory consumption. On average, the peak cache sizes of HDD+STR and Perses+STR are 18.06 GB and 145.56 MB, respectively. In the extreme case (*i.e.*, clang-27137), STR consume 98.67 GB memory in HDD and 698.91 MB in Perses. Such a large memory consumption poses a major concern for developers to adopt it in program reduction [37].

ZIP and SHA. The peak cache sizes of ZIP and SHA are much smaller than STR. On average, the peak cache sizes of ZIP in HDD and Perses are 2.95 GB and 24.55 MB, which are 84.34% and 82.99% smaller than STR, respectively. SHA further reduces the cache size to 27.70 MB and 679.3 KB in HDD and Perses. This is because ZIP and SHA store a compressed version of program variants, instead of their string representations.

RCC. RCC demonstrates the best memory efficiency by reducing the peak memory footprint to 1.01 MB and 50.5 KB in HDD and Perses. On average, compared to the second-best scheme SHA, the memory consumption of RCC is only 3.29% and 6.12% of the peak cache size of SHA in HDD and Perses, respectively. The reason is that RCC takes the advantage of compact encoding to reduce the size of each key in cache, and cache refresh to reduce the number of keys stored in cache. With such a small peak cache size, the overhead brought by RCC to system memory is negligible.

Table 2. The Statistics of the Subjects in Our Benchmark Suite. Column **Original Size** and **Reduction Size** shows the number of tokens before and after program reduction. Column **Query** refers to the number of property queries in program reduction. Column **Time** is the time of program reduction measured in seconds.

Subject	Original Size	HDD			Perses		
		Query	Time	Reduction Size	Query	Time	Reduction Size
clang-18556	27,511	175,197	9,515	439	3,266	4,012	227
clang-18596	44,097	226,832	12,904	425	3,589	4,095	266
clang-19595	31,280	163,032	9,677	230	2,435	3,626	156
clang-20680	48,173	293,610	17,295	379	6,052	6,890	549
clang-21467	28,283	337,278	17,089	432	4,698	5,741	177
clang-21582	38,964	323,280	17,202	1,067	9,005	4,458	649
clang-22337	70,770	262,252	10,998	564	4,906	1,355	268
clang-22382	21,068	276,550	6,566	194	3,019	348	144
clang-22704	184,444	187,773	9,078	81	2,359	992	78
clang-23309	38,647	448,300	22,234	1,035	6,233	1,239	464
clang-23353	30,196	369,063	11,970	143	3,324	439	98
clang-25900	78,960	307,042	10,378	462	3,200	547	239
clang-26350	123,811	352,659	44,698	429	4,008	2,377	195
clang-26760	209,577	200,694	19,096	303	2,765	1,141	116
clang-27137	174,538	715,750	172,058	531	5,823	4,828	180
clang-27747	173,840	95,849	2,914	332	2,077	626	117
clang-31259	48,799	330,658	17,971	590	3,560	1,969	406
gcc-59903	57,581	304,717	13,043	582	5,322	2,441	316
gcc-60116	75,224	302,558	13,455	1,304	6,331	1,798	488
gcc-60452	72,241	383,540	17,689	762	5,323	1,009	342
gcc-61047	17,179	202,335	7,687	556	2,450	753	270
gcc-61383	32,449	290,973	11,710	427	4,814	3,284	272
gcc-61917	85,359	234,941	11,010	232	3,746	816	150
gcc-64990	148,931	276,826	28,527	390	4,145	2,188	240
gcc-65383	43,942	255,402	9,619	234	2,724	782	153
gcc-66186	47,481	273,728	16,376	713	3,925	2,785	328
gcc-66375	65,488	353,404	29,482	856	4,854	3,251	440
gcc-66412	72,241	383,540	17,721	762	5,323	1,002	342
gcc-66691	20,044	301,720	12,597	949	6,937	3,296	587
gcc-70127	154,816	396,903	64,941	669	3,392	2,854	301
gcc-70586	212,259	382,999	72,787	967	5,523	4,846	159
Mean	79,942	303,529	23,816	550	4,359	2,445	281

To understand the scalability of each caching scheme. we further analyzed the correlation between the peak cache size and two properties of subjects, *i.e.*, the size (*i.e.*, number of tokens) of subjects and the number of property queries in program reduction. These two properties approximate the complexity of each subject in program reduction. Figure 2 plots the peak cache size of proposed caching schemes and the baseline *w.r.t.* the size and number of queries of each subject in HDD and Perses, respectively. STR and ZIP demand more memory when the size of input programs and the number of property queries increase, and thus they are not capable of handling large and complex subjects for program reduction. As shown in Figures 2c and 2d, the peak cache size of SHA is proportional to the number of queries, since the number of keys in SHA increases when the number of property queries increases and each key has the same size in memory. By contrast, since the number of keys in RCC is regularly reduced by the cache refreshing and each key has a small variable size, the peak cache size of RCC does not always increase when input program

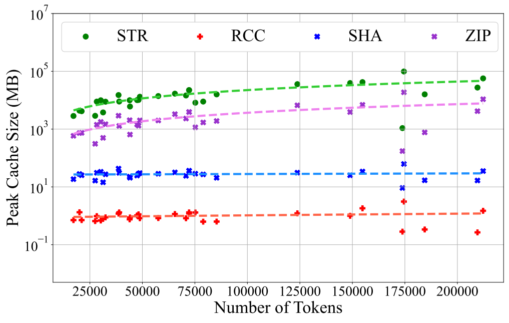
Table 3. Peak Cache Size (KB) in HDD and Perses

Subject	HDD				Perses			
	STR	ZIP	SHA	RCC	STR	ZIP	SHA	RCC
clang-18556	2,884,142	310,475	16,494	656	32,793	4,509	474	41
clang-18596	6,057,853	648,072	20,743	876	55,051	7,605	565	43
clang-19595	3,778,855	498,398	14,413	780	43,503	6,483	349	27
clang-20680	10,226,005	1,317,369	28,592	1,122	81,345	13,400	1,204	92
clang-21467	8,952,624	1,420,649	30,863	974	58,724	11,026	709	35
clang-21582	9,004,373	1,293,164	30,565	1,311	109,522	19,949	1,511	119
clang-22337	14,589,500	2,322,056	24,181	824	141,930	23,631	787	44
clang-22382	4,168,392	743,016	25,280	709	34,340	6,644	484	60
clang-22704	15,967,138	769,719	16,927	336	259,634	14,171	343	46
clang-23309	15,073,216	2,910,031	42,996	1,191	96,980	20,480	965	60
clang-23353	9,934,809	1,777,326	33,409	688	85,239	15,976	582	99
clang-25900	8,976,320	1,700,111	27,058	627	104,835	18,551	510	34
clang-26350	35,923,321	6,665,960	31,140	1,218	289,411	55,645	591	20
clang-26760	27,386,315	4,182,515	16,662	267	221,971	34,509	430	84
clang-27137	98,674,879	18,762,875	62,193	3,106	698,906	138,850	938	26
clang-27747	1,083,924	173,705	9,188	283	22,151	3,767	323	44
clang-31259	13,161,378	2,009,604	30,050	825	83,825	14,143	493	37
gcc-59903	13,926,538	1,989,642	28,520	834	152,856	24,603	853	22
gcc-60116	8,193,094	1,151,006	28,684	1,302	141,402	22,593	953	33
gcc-60452	22,320,641	3,898,990	36,189	1,208	159,600	28,716	851	25
gcc-61047	2,851,067	585,659	18,526	708	23,959	5,005	403	70
gcc-61383	8,875,442	1,479,231	27,309	881	72,506	13,594	693	59
gcc-61917	15,909,599	1,910,068	20,693	632	125,993	17,633	569	87
gcc-64990	39,219,366	3,886,376	25,217	1,002	278,744	30,772	610	46
gcc-65383	9,982,340	2,038,062	22,617	755	66,015	13,318	397	28
gcc-66186	10,103,904	1,445,763	24,677	1,027	67,396	10,659	549	38
gcc-66375	16,855,199	3,297,347	31,815	1,154	105,141	20,347	648	26
gcc-66412	22,320,641	3,911,712	36,189	1,311	159,600	28,716	851	59
gcc-66691	4,359,484	731,768	27,972	1,327	60,555	11,956	985	78
gcc-70127	42,270,240	6,927,089	34,073	1,819	257,950	42,895	556	52
gcc-70586	56,885,504	10,816,579	35,332	1,490	420,536	81,023	885	29
Mean	18,061,809.7	2,954,010.8	27,695.7	1,007.9	145,561.7	24,553.9	679.3	50.4
Relative Diff. *w.r.t.* STR	0%	84.34%	99.72%	99.99%	0%	82.99%	99.23%	99.93%
Ratio *w.r.t.* RCC	18,605.68	2,808.75	30.38	1	4,097.68	714.02	16.35	1

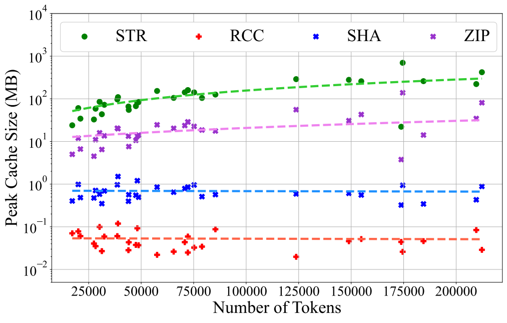
[1] Relative Diff. *w.r.t.* STR : (STR − [Caching Scheme]) ÷ STR × 100%.
[2] Ratio *w.r.t.* RCC : [Caching Scheme] ÷ RCC.

becomes more complex. As shown in Figure 2, RCC has a much flatter curve than others, and its cache size barely increases when the input program size scales up (*e.g.*, 10 folds, from 17,179 to 212,259 tokens) and the number of queries increases.

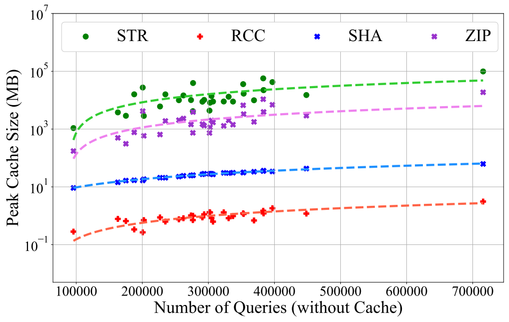
Finding 1.1: Compared to STR, the three caching schemes proposed in this study, *i.e.*, ZIP, SHA and RCC, are effective to reduce the peak cache size in both HDD and Perses. RCC has the smallest peak cache size in all 31 subjects, which is 96.4% and 91.74% smaller than the second-best caching scheme in HDD and Perses.



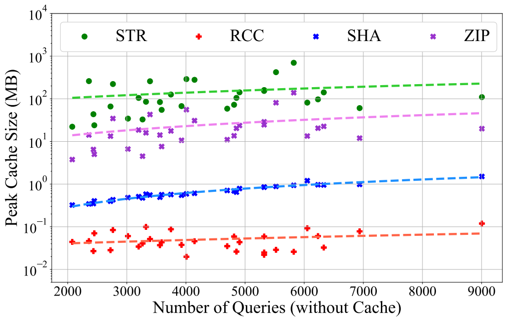
(a) Peak Cache Size vs #Tokens in HDD



(b) Peak Cache Size vs #Tokens in Perses



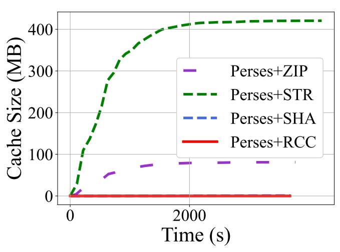
(c) Peak Cache Size vs #Queries in HDD



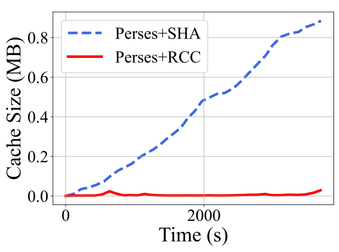
(d) Peak Cache Size vs #Queries in Perses

Fig. 2. Scalability of Caching Scheme on 31 Subjects in HDD and Perses. Peak cache size (y-axis) is in log scale.

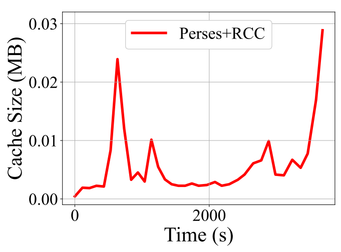
5.2.2 Average Cache Size. To understand the memory footprint in the entire reduction process, we further profiled the cache size of each caching scheme over time. Figure 3 visualizes the memory consumption of caching schemes in Perses over time on gcc-70586 (the subject with the most tokens, *i.e.*, 212,159). The memory consumption of STR, ZIP and SHA all accumulate over time, as shown in Figures 3a and 3b. In contrast, Figure 3c shows that the cache size of RCC not only always retains at a low level, but also does not monotonically increase along with time. This is because RCC regularly refreshes the cache to remove the program variants that will not be generated in the subsequent reduction.



(a) Overview



(b) SHA vs RCC



(c) Zoom-in detail of RCC

Fig. 3. Memory Consumption over Time on subject gcc-70586.

Table 4. Average Cache Size (KB) of SHA and RCC in Perses. The number in the parentheses is the ratio of the average cache size of RCC *w.r.t.* the average ratio of SHA.

Subject	SHA	RCC	Subject	SHA	RCC
clang-18556	274.10	9.70 (3.5%)	gcc-59903	372.38	8.29 (2.2%)
clang-18596	296.05	11.45 (3.9%)	gcc-60116	524.40	14.97 (2.9%)
clang-19595	271.34	11.03 (4.1%)	gcc-60452	287.66	5.61 (2.0%)
clang-20680	516.68	12.80 (2.5%)	gcc-61047	184.39	10.77 (5.8%)
clang-21467	288.62	5.37 (1.9%)	gcc-61383	369.21	11.55 (3.1%)
clang-21582	516.68	12.80 (2.5%)	gcc-61917	200.12	3.71 (1.9%)
clang-22337	302.93	5.78 (1.9%)	gcc-64990	252.94	7.43 (2.9%)
clang-22382	318.63	3.68 (1.2%)	gcc-65383	177.39	3.41 (1.9%)
clang-22704	112.33	4.16 (3.7%)	gcc-66186	266.50	10.79 (4.0%)
clang-23309	363.22	7.12 (2.0%)	gcc-66375	325.71	15.87 (4.9%)
clang-23353	270.58	9.26 (3.4%)	gcc-66412	287.96	6.24 (2.2%)
clang-25900	202.17	6.06 (3.0%)	gcc-66691	567.13	17.31 (3.1%)
clang-26350	252.16	7.15 (2.8%)	gcc-70127	289.27	10.66 (3.7%)
clang-26760	138.70	3.11 (2.2%)	gcc-70586	630.92	5.96 (0.9%)
clang-27137	337.49	7.44 (2.2%)			
clang-27747	142.11	4.50 (3.2%)			
clang-31259	417.61	10.67 (2.6%)			
Mean Ratio of RCC *w.r.t.* SHA: 2.8%					

We further measured the average cache sizes of SHA and RCC in Perses, since they have significant advantages over STR and ZIP. Table 4 shows the results. On average, the average cache size of RCC is only 50.5 KB while the average cache size of SHA is 679.3 KB. In other words, the average cache size of RCC is only 2.8% (0.9% ~ 2.8%) of the one of SHA. Such advantages come from the two key features of RCC, *i.e.*, compact encoding and cache refresh. The former minimizes the size of each cache key and the latter reduces the number of keys.

Finding 1.2: Throughout the entire program reduction process, RCC maintains a consistently low memory consumption ($\leq$ 20 KB), which is 96.4% and 91.74% smaller than the second-best caching scheme in HDD and Perses.

Answer to RQ1: While all proposed caching schemes improve memory efficiency, RCC outperforms others by at least one magnitude, in terms of both peak cache size and average cache size.

5.3 RQ2: Program Reduction Efficiency

We evaluated the program reduction efficiency of the proposed caching schemes by measuring the number of queries and reduction time respectively. Table 5 shows the information on both queries and reduction time of different caching schemes in HDD and Perses.

5.3.1 Number of Queries. In program reduction, it may take considerable time to execute a property query and a typical program reduction process can have thousands of queries. Thus, it is common to use the number of queries to measure the program reduction efficiency [17, 34, 43, 52]. In this section, we aim to study whether the proposed schemes can effectively avoid the redundant queries in program reductions.

Table 5. Program Reduction Efficiency Comparison. All time is measured in seconds. Columns STR, ZIP, SHA, RCC show the reduction time(in seconds) using each cache scheme respectively.

Subject	HDD							Perses						
	No Caching		Caching					No Caching		Caching				
	Query	Time	Query	STR	ZIP	SHA	RCC	Query	Time	Query	STR	ZIP	SHA	RCC
clang-18556	175,197	9,515	69,438	7,755	7,925	7,766	7,751	3,266	4,012	2,331	3,934	3,942	3,937	3,927
clang-18596	226,832	12,904	88,618	10,519	10,917	10,530	10,508	3,589	4,095	2,797	3,987	3,997	3,991	3,981
clang-19595	163,032	9,677	60,139	7,987	8,283	7,984	7,969	2,435	3,626	1,805	3,552	3,560	3,554	3,544
clang-20680	293,610	17,295	111,317	14,247	14,871	14,243	14,140	6,052	6,890	4,228	6,235	6,373	6,230	6,193
clang-21467	337,278	17,089	129,033	13,470	14,416	13,515	13,444	4,698	5,741	3,545	5,670	5,677	5,665	5,656
clang-21582	323,280	17,202	127,888	12,671	13,409	12,721	12,644	9,005	4,458	6,745	3,264	3,288	3,269	3,256
clang-22337	262,252	10,998	99,207	8,130	9,597	8,199	8,113	4,906	1,355	3,783	1,192	1,226	1,206	1,190
clang-22382	276,550	6,566	104,047	3,942	4,263	3,950	3,932	3,019	348	2,380	303	307	301	287
clang-22704	187,773	9,078	71,339	7,529	8,097	7,337	7,175	2,359	992	1,811	814	823	812	795
clang-23309	448,300	22,234	183,365	16,196	17,558	16,101	15,846	6,233	1,239	4,602	980	995	979	971
clang-23353	369,063	11,970	140,447	8,414	9,295	8,515	8,298	3,324	439	2,812	401	407	399	389
clang-25900	307,042	10,378	111,939	7,274	8,152	7,233	7,196	3,200	547	2,500	477	493	474	464
clang-26350	352,659	44,698	130,430	41,284	47,433	41,298	40,820	4,008	2,377	3,088	2,127	2,172	2,123	2,105
clang-26760	200,694	19,096	70,292	17,287	20,109	17,286	16,435	2,765	1,141	2,183	1,030	1,059	1,031	1,018
clang-27137	715,750	172,058	260,138	162,080	176,050	162,671	161,709	5,823	4,828	4,911	4,225	4,326	4,217	4,210
clang-27747	95,849	2,914	39,065	1,866	1,971	1,860	1,856	2,077	626	1,660	548	553	545	530
clang-31259	330,658	17,971	125,359	13,702	14,804	13,571	13,417	3,560	1,969	2,418	1,136	1,144	1,130	1,119
gcc-59903	304,717	13,043	118,605	10,203	11,439	9,977	9,857	5,322	2,441	4,180	2,170	2,185	2,132	2,121
gcc-60116	302,558	13,455	119,412	9,741	10,263	9,659	9,651	6,331	1,798	4,556	1,323	1,333	1,316	1,309
gcc-60452	383,540	17,689	153,010	13,602	15,773	13,398	13,372	5,323	1,009	4,178	886	918	894	885
gcc-61047	202,335	7,687	78,590	4,847	5,171	4,869	4,845	2,450	753	1,935	574	588	583	569
gcc-61383	290,973	11,710	113,247	8,771	9,649	8,663	8,594	4,814	3,284	3,475	2,197	2,208	2,195	2,185
gcc-61917	234,941	11,010	88,297	8,638	9,817	8,780	8,596	3,746	816	2,936	723	735	718	713
gcc-64990	276,826	28,527	103,848	25,413	28,596	26,170	25,411	4,145	2,188	3,086	1,925	1,959	1,917	1,908
gcc-65383	255,402	9,619	96,839	7,015	8,087	7,165	6,967	2,724	782	2,002	650	665	643	635
gcc-66186	273,728	16,376	101,388	12,733	13,328	12,595	12,541	3,925	2,785	2,666	1,892	1,891	1,881	1,865
gcc-66375	353,404	29,482	133,296	24,126	25,993	24,130	23,770	4,854	3,251	3,137	1,926	1,947	1,930	1,922
gcc-66412	383,540	17,721	153,010	13,698	15,988	13,715	13,534	5,323	1,002	4,177	886	915	894	883
gcc-66691	301,720	12,597	116,175	8,934	9,276	8,922	8,895	6,937	3,296	4,640	2,042	2,048	2,039	2,029
gcc-70127	396,903	64,941	143,553	58,323	63,526	57,810	57,648	3,392	2,854	2,771	2,153	2,180	2,147	2,135
gcc-70586	382,999	72,787	149,319	67,642	73,292	65,952	65,937	5,523	4,846	4,451	3,757	3,812	3,745	3,733
Mean	303,529	23,816	115,827	20,259	22,173	20,212	20,038	4,359	2,445	3,284	2,032	2,056	2,029	2,017

As aforementioned, program reducers like HDD and Perses issue redundant queries. As per numbers in Table 5, caching effectively reduces the number of queries issued in HDD by 61.8% from 303,529 (column 2) to 115,827 (column 4). Meanwhile, Perses with caching only issues 3,284 (column 15) queries while Perses along issues 4,359 (column 11). In conclusion, caching is effective in reducing the number of queries in HDD and Perses.

Notice that with STR, ZIP, SHA or RCC, HDD and RCC always issue the same number of queries, as shown in Table 5; this consistency reveals the correctness of each scheme. It is worth noting that RCC will not cause extra queries even though it refreshes the cache periodically. This result confirms the safety of cache refreshing in RCC, *i.e.*, the removed programs will not be generated in the remaining program reduction process, formally proved in §4.3.4.

Finding 2.1: On average, caching can avoid 61.8% and 24.3% of property queries in HDD and Perses, respectively.

5.3.2 *Reduction Time.*
Table 5 shows the runtime of program reduction with each caching scheme and without caching. We focus on the discussion from the following two perspectives.

With RCC* v.s. *Without Caching. Comparing to program reduction without caching, RCC reduces the time by 22.8% in HDD and 18.2% in Perses. Specifically, the average program reduction runtime in HDD without caching is approximately 6.6 hours (23,816 seconds in column 3, Table 5), while RCC shortens the average reduction time to around 5.6 hours (20,038 seconds). In the best cases, RCC shortens the reduction time by 40.1% and 43.2% in HDD and Perses, respectively. Such an improvement in efficiency facilitates the debugging process and saves the time and computation resources for software developers.

We believe that such a speed-up is significant from four perspectives. First, many researches in compiler testing, such as CSmith [51], YARPGen [31] and EMI [26, 27, 41], identified hundreds or thousands of bugs in C compilers. Given the large number of input programs to be reduced, saving around 20% of the time in program reduction can greatly facilitate the workflow of testing and debugging. Second, in industry, program reduction is one of the daily routines of automated testing. Developers rely on program reduction to reduce interesting programs found by automated test techniques such as fuzzing. Reducing the time in program reduction up to 43.2% can improve the productivity of software developers. Moreover, program reduction also has broad applications in other domains [11], such as program optimization [39] and program understanding [3], meaning that users in these fields can also benefit from the increased efficiency in program reduction. Lastly, shorter runtime is preferred in society, aligning with the principles of green and sustainable computing.

RCC* v.s. *Others. In HDD, RCC, SHA and STR have similar performance in terms of runtime. RCC has the shortest runtime which is 1.07% faster than HDD+SHA. We notice that ZIP runtime performance is slower than others in HDD and we conjecture that the compression process of ZIP is more time-consuming than others. In Perses, the performance of all four caching schemes are statistically comparable with each other, but RCC still surpasses the baseline method SHA by a small fraction (1.05%). Such results demonstrate the efficiency of RCC in encoding and refreshing. Please note that although STR does not spend time in encoding, comparison between strings usually take more time than one between the compact encoding of RCC, since the string representation of a program usually has a larger size than compact encoding.

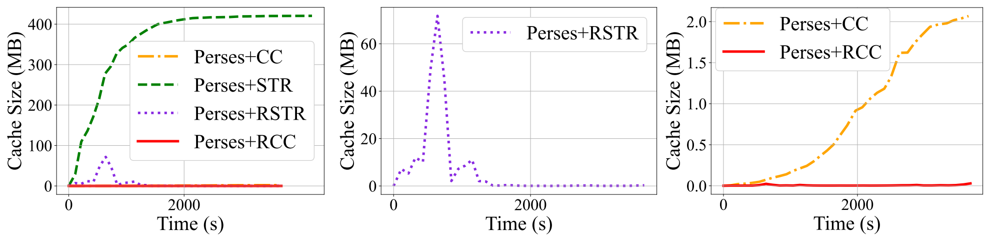
Finding 2.2: All caching schemes shorten the runtime to a similar extent, except ZIP in HDD. Perses is slightly faster than the others by around 1%.

Answer to RQ2: Caching schemes can effectively avoid redundant queries, by 61.8% in HDD and 24.3% in Perses, and shorten the reduction time up to 40.1% and 43.2%, respectively. Perses outperforms the others by around 1% in terms of runtime.

5.4 RQ3: Effects of Compact Encoding and Cache Refreshing

To understand the individual effect of compact encoding and refreshable caching in RCC, we conducted the following ablation study.

Compact Encoding. We constructed a variant caching scheme, CC, in Perses and measured its peak cache size during the program reduction process (column 2 in Table 6). **Perses+CC** is a variant of Perses+STR by replacing the string-based encoding with the compact encoding proposed by us. It can also be viewed as a variant of Perses+RCC by disabling the cache refreshing. The programs



(a) Overview (b) Perses+RSTR (c) Perses+RCC and Perses+CC

Fig. 4. Memory Consumption over Time on subject gcc-70586. Figure 4a compares on memory consumption over Time on subject gcc-70586. Figures 4b and 4c show a zoomed-in view of Perses+RSTR, Perses+RCC and Perses+CC.

added into the cache will never be removed, and the encoding of each program is always computed w.r.t. the input program.

Compact encoding leads to a minimal peak cache size. Figure 4a shows the memory consumption of Perses+CC is only a fraction of Perses+STR. Perses+CC significantly reduces the memory footprint (99.5% averagely). However, as shown in Figure 4c, without cache refreshing, the memory footprint of Perses+CC accumulates over time and increases to 2 MB eventually. In other words, the Compact Encoding is an effective compression technique that can reduce the size of cache, while it cannot prevent the increase of the cache size over time.

Refreshable Caching. Similarly, we constructed a variant caching scheme, **Perses+RSTR** (see column 3 in Table 6) by adding cache refreshing capability to Perses+STR. Instead of storing the string of the source code as the cache key to the cache, we choose to add the list of program tokens, so that Perses+RSTR can restore these variants from the cache keys and perform cache refreshing effectively.

Figure 4a illustrates the effect of cache refreshing by comparing Perses+RSTR with Perses+STR. As unnecessary entries in the cache are removed when new min programs are found, Perses+RSTR effectively reduces the peak cache size by 93.8% on average when compared to Perses+STR. In summary, refreshable cache ensures that the cache contains only the necessary elements during the program reduction process, resulting in relatively small memory footprint. However, since the entry of RSTR is in the form of strings, instead of compact encoding, the entire cache size is much larger than RCC, especially at the beginning of program reduction process.

Answer to RQ3: Both compact encoding and cache refreshing are effective in improving memory efficiency. Compact encoding minimizes the memory footprint of each cache key, and cache refreshing removes stale cache keys in time to further minimize the whole memory footprint of the cache.

6 DISCUSSION

6.1 Caching for Delta Debugging

We also studied the impact of caching on DD. However, DD is not as good at reducing structured inputs, *e.g.*, programs, as HDD and Perses; it issues more queries and takes much more time to reduce a benchmark subject. Due to time limits, we could only finish a similar experiment on one small subject, gcc-71626 (6,133 tokens). Table 7 shows the statistics.

Table 6. Peak Cache Size (KB) of CC and RSTR

	Peak Cache Size in Perses			
Subject	STR	RSTR	CC	RCC
clang-18556	32,793	5,138	754	41
clang-18596	55,051	18,384	1,044	43
clang-19595	43,503	10,981	495	27
clang-20680	81,345	22,107	3,052	92
clang-21467	58,724	15,784	1,320	35
clang-21582	109,522	20,305	4,001	119
clang-22337	141,930	52,142	1,686	44
clang-22382	34,340	257	708	60
clang-22704	259,634	63	520	46
clang-23309	96,980	2,231	2,164	60
clang-23353	85,239	112	734	99
clang-25900	104,835	623	713	34
clang-26350	289,411	1,987	1,293	20
clang-26760	221,971	122	672	84
clang-27137	698,906	100,835	2,643	26
clang-27747	22,151	153	456	44
clang-31259	83,825	1,578	756	37
gcc-59903	152,856	3,387	1,776	22
gcc-60116	141,402	2,400	2,025	33
gcc-60452	159,600	54,621	1,838	25
gcc-61047	23,959	4,362	588	70
gcc-61383	72,506	718	1,345	59
gcc-61917	125,993	229	1,086	87
gcc-64990	278,744	4,077	1,303	46
gcc-65383	66,015	220	523	28
gcc-66186	67,396	1,066	944	38
gcc-66375	105,141	1,950	1,250	26
gcc-66412	159,600	58,656	1,838	59
gcc-66691	60,555	7,065	2,478	78
gcc-70127	257,950	918	929	52
gcc-70586	420,536	26,211	2,068	29
Mean	145,562	13,506	1,387	50
Relative Diff. *w.r.t.* STR	0.00%	89.583%	98.543%	99.932%
Ratio *w.r.t.* RCC	4,097.7	381.1	33.2	1.0

[1] Relative Diff. *w.r.t.* STR : (STR − [Caching Scheme]) ÷ STR × 100%.
[2] Ratio *w.r.t.* RCC : [Caching Scheme] ÷ RCC.

Without caching, DD issues more than five millions queries and takes 29.7 hours to find the 1-minimal output. With RCC, DD only issued 1.5 millions queries, which is 73.0% improvement. The overall time is significantly reduced to around 9.8 hours, which is 67.1% faster. Further, RCC outmatches STR in DD in terms of time and memory footprint. Compared to STR (17.6 GB), RCC takes only 208 KB. This result further demonstrates that RCC is a general approach for deletion-based program reduction algorithms.

6.2 Sized-Based Refreshing

An alternative cache-refresh algorithm for RCC is to record the size of each variant program and remove any programs of which the sizes are equal to or larger than min from cache. This is

Table 7. Comparison of STR and RCC on DD.

	DD	DD+STR	DD+RCC
Query	5,477,887	1,480,695	1,480,695
Time (s)	106,857	35,296	35,167
Cache Size (KB)	N/A	17,641,803	208

because the program reduction process starting from min will not generate any variant programs that are larger in size than min. We refer to this alternative as size-based refreshing and name the implementation as $\mathsf{RCC}_{\mathsf{size}}$. Algorithm 5 details the sized-based refreshing in $\mathsf{RCC}_{\mathsf{size}}$. On line 5, $\mathsf{RCC}_{\mathsf{size}}$ compares the size of program p with the size of min and only keeps the programs of which the sizes are smaller than than min.

Algorithm 5: Size-Based Refreshing in $\mathsf{RCC}_{\mathsf{size}}$

```
1 Function SizeBasedRefreshCache(old_cache, prev, min):
      Input: old_cache: the cache used previously
      Input: prev: the previous min
      Input: min: the current/new min
2     cache ← ∅
3     for encoding ∈ old_cache do
4         p′ ← CompactDecode(prev, encoding)
5         if |p′| ≥ |min| then continue
6         cache ← cache ∪ {CompactEncode(min, p′)}
7     return cache
```

Conceptually, the cache entries evicted by RCC are a superset of those by $\mathsf{RCC}_{\mathsf{size}}$, because $|p'| \geq |\mathsf{min}|$ is just one of the multiple sufficient conditions for $p' \not\sqsubseteq \mathsf{min}$. Specifically, RCC removes the variant program entries from cache that cannot be derived from min in the rest of the program reduction process; this process removes not only all the programs that have equal or larger size than min, but also any programs that have smaller sizes than min and are not proper subsequences of min.

To demonstrate the benefit of RCC over $\mathsf{RCC}_{\mathsf{size}}$, Figure 5 shows the memory footprint and the cache entry count on subject clang-26760 in HDD and Perses. In terms of memory footprint, the gap between RCC and $\mathsf{RCC}_{\mathsf{size}}$ widens over time, especially towards the end of the program reduction process. As for the cache key count, RCC constantly has noticeably fewer cache key entries than $\mathsf{RCC}_{\mathsf{size}}$ throughout the process in both HDD and Perses. RCC removes more cache key entries and only has approximately 40% fewer entries than $\mathsf{RCC}_{\mathsf{size}}$ in HDD. Furthermore, size-based refreshing appears less effective in removing cache key entries when the size of min is small, since the cache key entry count soars near the end of the process in both HDD and Perses.

6.3 Limitation and Future Work

6.3.1 Targeted Program Reduction Algorithms. The proposed caching schemes are agnostic to most program reduction algorithms. The majority of these algorithms employ solely deletion-based program transformations [16, 17, 34, 43, 52], which are well-supported by RCC. Language-specific algorithms such as, C-Reduce [37], Jax [47] and JShrink [4], may contain non-deletion-based transformations, which are usually specific to certain languages like C or Java. In RCC, if a non-deletion-based transformation generates a variant, the variant can be processed as a cache miss

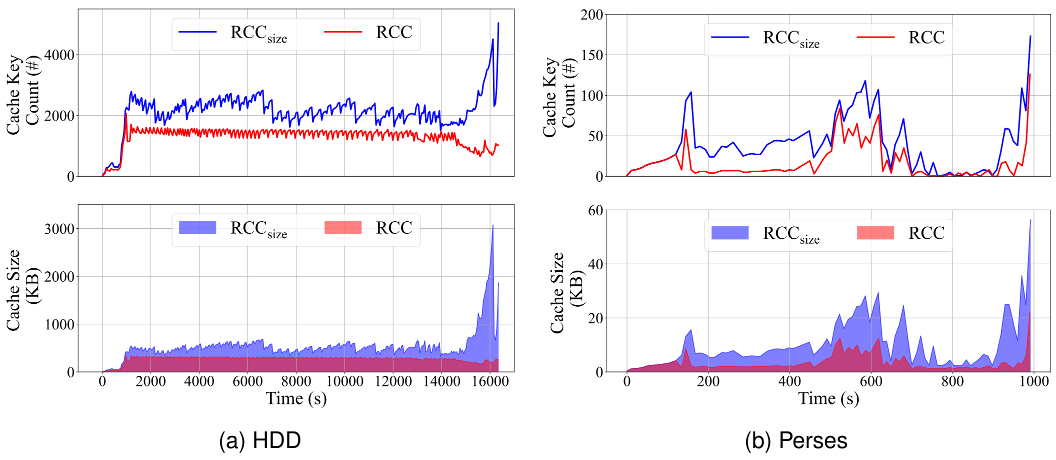


Fig. 5. Comparison between RCC and RCC$_{size}$ in terms of Memory Footprint (Line) and Cache Key Count (Area) over Time on subject clang-26760.

due to no available encoding, resulting in a property test. However, this cache miss does not affect the correctness of the reduction process.

Note that the cache-miss scenario due to non-deletion-based transformation is still rare. For example, in C-Reduce, the majority of its program transformations are still language-agnostic deletion-based, and only a minor fraction of its program transformations are non-deletion-based. We investigated the program reduction process using C-Reduce on clang-22704, and we found that 8807 out of 8963 (98.3%) transformations are generic and deletion-based; 156 transformations (1.7%) leverage C-specific knowledge, and quite a few of them are still deletion-based. Therefore, though RCC currently does not support non-deletion-based transformations, the amount of redundant queries caused by non-deletion-based transformations are likely to be marginal. As followup work, we are making efforts to accommodate such non-deletion-based transformations.

6.3.2 Benchmark Suite. The subjects used in our benchmark suite may not cover all possible programming languages and various bugs for program reduction. We followed the previous studies in program reduction [25, 38, 43, 50, 53] and used the same benchmark in Perses [43]. Moreover, we collected 11 more from the official GCC and LLVM bug repositories. In total, our benchmark suite contains 31 C compiler bugs, which consists of programs that have the common size of automatically generated files via fuzzing techniques such as CSmith [51] and EMI [26]. These bugs cover both medium-scale and large-scale compiler bugs. To our best knowledge, this is the largest benchmark suite from real C compiler bug repositories.

6.3.3 Language-agnostic Caching Scheme. Furthermore, the proposed caching schemes have no assumption on the language of the program to be reduced and does not have any language-specific optimizations. Even though we used the C/C++ programs in the evaluation, ZIP, SHA and RCC are general caching schemes that can be used in the program reduction of other programming languages.

6.3.4 Threats to Validity. A possible threat to validity is the implementation of RCC. To mitigate this threat, we took two measures. First, we used intensive assertions and wrote test cases to validate the correctness of computation steps in RCC, especially encoding and decoding. Second, the implementation was reviewed by multiple authors to avoid possible mistakes.

Another threat is the measurement of memory efficiency (*i.e.*, cache size) and program reduction efficiency (*i.e.*, time and number of queries). For cache size, we adopted a third-party library to measure the size of each caching scheme. Specially, the cache size includes the size of object that stores cache entries and the size of all cache entries. The objects other than these objects in JVM, such as the objects created for memory profiling, are explicitly excluded. As for time, all experiments were conducted in a separate cloud virtual machine multiple times to avoid possible fluctuations due to background tasks. For fair evaluations, we disabled assertions when measuring the time of each caching scheme. In terms of the number of queries, we checked the number of queries across different caching schemes and we did not find any case where the numbers of queries vary in different caching schemes. Both the implementation and evaluation scripts are publicly available for reproducibility and replicability at https://github.com/uw-pluverse/perses/tree/master/doc/RCC.md

7 RELATED WORK

We survey three lines of the closely related work.

7.1 Caching in Program Reduction

Hodován *et al.* [20] is the first literature on program reduction that formally presented the idea of test outcome caching. Based on the observation that the different configuration yields the same variant from time to time during the process, it leveraged the string-based caching approach to store the pairs of the current best programs and their corresponding test outcomes. Similarly, C-Reduce, a highly customized program reduction tool for C/C++, employed a simple string-based cache approach at the level of passes to store the entire current best program [38].

As we discussed and evaluated in previous sections, STR scheme has larger overheads and poor scalability. In contrast, ZIP and SHA are memory-efficient; especially, RCC presents a fresh way for efficient caching while offering enormous scaling potential.

7.2 General Caching Algorithms

Caching is widely applied to software systems [5, 6, 30], *e.g.*, caching contents in networks for better user experience. In general, an application stores either prefethced data or pre-computed results into a cache to facilitate the execution. To be cost-effective and to enable efficient use of data, caches must be relatively small [15]. The general caching algorithm leverages the locality of references, because temporal and spatial locality hint the likelihood of data to be accessed next. When the cache is full, the algorithm must choose which items to discard to make room for new ones; cache eviction algorithms aim to keep the cache a constant, compact size. Classical algorithms include LRU [21, 35] and MRU [8, 9].

In the setting of program reduction, locality of references does not work well because temporal locality rarely shows in program reduction. Furthermore, given the negligible memory overhead of RCC, a program reduction algorithm equipped with RCC does not need the classical cache eviction algorithms such as LRU and MRU to mitigate memory overhead.

7.3 Optimization for Program Reduction

There have been a great number of the program reduction techniques [4, 17, 19, 25, 43–45, 48, 52, 53]. Besides the cache, researchers also proposed other methods to improve the performance of program reduction in diverse ways. For example, Hodován *et al.* [20] proposed two optimization techniques as the pre-processing, including vertical tree squeezing and unresolvable tokens hiding, in order to speed up program reduction. Kalhauge *et al.* introduced J-Reduce for Java bytecode reduction [22], and recently they further reduced bytecode via propositional logic [23].

All proposed caching schemes belong to the same category of performance optimization of program reduction. They provides a memory-efficient cache for program reduction, which is orthogonal to other optimization techniques.

8 CONCLUSION

This paper is the first effort to conduct systematic, extensive analysis of memory-efficient caching schemes for program reduction. We introduce three effective schemes; two exploit readily available compression libraries, namely ZIP and SHA. We also present a novel, domain-specific caching scheme RCC to empower the program reduction by compact encoding and refreshable caching. Our extensive evaluation on 31 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 61.8% and boost the runtime performance by 22.8%. For memory efficiency, caching schemes ZIP and SHA cut down the memory overhead by 84.34% and 99.72%, compared to the state-of-the-art STR; furthermore, the highly-scalable, domain-specific RCC dominates peer schemes, and outperforms the second-best SHA by remarkably 96.4%. As generic, language-agnostic caching schemes, ZIP, SHA and RCC are readily applicable to program reduction techniques and facilitate program reduction algorithms.

The implementation of all caching schemes is publicly available at https://github.com/uw-pluverse/perses/tree/master/doc/RCC.md. Moreover, RCC has been enabled by default in Perses, because of its efficiency and advantageously low memory footprint compared to the others.

ACKNOWLEDGMENTS

We thank the associated editor and anonymous reviewers of TOSEM for their valuable feedback. Their insightful suggestions helped us improve this manuscript. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, a project under Waterloo-Huawei Joint Innovation Lab, and CFI-JELF Project #40736. We express our gratitude to Microsoft for generously providing Azure credits for this research.

REFERENCES

[1] Dimitris Andreour. 2015. ObjectExplorer. Retrieved May 29, 2023 from https://github.com/DimitrisAndreou/memory-measurer

[2] Antoine Balestrat. 2012. CCG: A Random C Code Generator. Retrieved May 29, 2023 from https://github.com/Merkil/ccg/

[3] David W. Binkley, Nicolas Gold, Mark Harman, Syed S. Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 109–120. https://doi.org/10.1145/2635868.2635893

[4] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: in-depth investigation into debloating modern Java applications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 135–146. https://doi.org/10.1145/3368089.3409738

[5] Federico Brunero and Petros Elia. 2023. Fundamental Limits of Combinatorial Multi-Access Caching. *IEEE Transactions on Information Theory* 69, 2 (2023), 1037–1056. https://doi.org/10.1109/TIT.2022.3193723

[6] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. 2009. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*. IEEE Computer Society, 462–469. https://doi.org/10.1109/ICPP.2009.22

[7] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 197–208. https://doi.org/10.1145/2491956.2462173

[8] Hong Tai Chou and David J. DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica* 1, 1 (01 Nov 1986), 311–336. https://doi.org/10.1007/BF01840450

[9] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 330–341. http://www.vldb.org/conf/1996/P330.PDF

[10] Peter Deutsch and Jean-Loup Gailly. 1996. RFC1950: ZLIB Compressed Data Format Specification Version 3.3. (1996). https://doi.org/10.17487/RFC1950

[11] Alastair Donaldson and David MacIver. 2021. *Test Case Reduction: Beyond Bugs.* Retrieved May 29, 2023 from https://blog.sigplan.org/2021/05/25/test-case-reduction-beyond-bugs

[12] GCC. 2017. *A Guide to Testcase Reduction.* Retrieved May 29, 2023 from https://gcc.gnu.org/wiki/A_guide_to_testcase_reduction

[13] Mark Adler Greg Roelofs. 1996. *A Massively Spiffy Yet Delicately Unobtrusive Compression Library.* Retrieved Jul 1, 2023 from https://zlib.net

[14] Shay Gueron, Simon Johnson, and Jesse Walker. 2011. SHA-512/256. In *Eighth International Conference on Information Technology: New Generations, ITNG 2011, Las Vegas, Nevada, USA, 11-13 April 2011*, Shahram Latifi (Ed.). IEEE Computer Society, 354–358. https://doi.org/10.1109/ITNG.2011.69

[15] Jim Handy. 1998. *The Cache Memory Book (2nd Ed.): The Authoritative Reference on Cache Design.* Academic Press, Inc., USA.

[16] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 380–394. https://doi.org/10.1145/3243734.3243838

[17] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 861–871. https://doi.org/10.1109/ASE.2017.8115697

[18] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya (Eds.). ACM, 31–37. https://doi.org/10.1145/2994291.2994296

[19] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse Hierarchical Delta Debugging. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 194–203. https://doi.org/10.1109/ICSME.2017.26

[20] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. (2017), 23–29. https://doi.org/10.1109/AST.2017.4

[21] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 439–450. http://www.vldb.org/conf/1994/P439.PDF

[22] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 556–566. https://doi.org/10.1145/3338906.3338956

[23] Christian Gram Kalhauge and Jens Palsberg. 2021. Logical bytecode reduction. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1003–1016. https://doi.org/10.1145/3453483.3454091

[24] J. Kelsey, S. Change, R. Perlner, Information Technology Laboratory (National Institute of Standards, and Technology). 2016. *SHA-3 Derived Functions: CSHAKE, KMAC, TupleHash and ParallelHash.* Vol. 800. U.S. Department of Commerce, National Institute of Standards and Technology. 185 pages.

[25] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. HDDr: a recursive variant of the hierarchical Delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 16–22. https://doi.org/10.1145/3278186.3278189

[26] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[27] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages,*

and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[28] Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI, Article 180 (jun 2023), 25 pages. https://doi.org/10.1145/3591294

[29] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 65–76. https://doi.org/10.1145/2737924.2737986

[30] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 795–809. https://doi.org/10.1145/3037697.3037731

[31] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

[32] LLVM. 2017. *How to submit an LLVM bug report.* Retrieved Mar 20, 2023 from https://llvm.org/docs/HowToSubmitABug.html

[33] LLVM/Clang. 2022. Clang documentation – LibTooling. Retrieved May 29, 2023 from https://clang.llvm.org/docs/LibTooling.html

[34] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical Delta Debugging. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

[35] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm For Database Disk Buffering. (1993), 297–306. https://doi.org/10.1145/170035.170081

[36] John Regehr. 2015. *Reducers are Fuzzers – EMBEDDED IN ACADEMIA.* Retrieved Jul 1, 2023 from https://blog.regehr.org/archives/1284

[37] John Regehr. 2016. [creduce-dev] cache. Retrieved May 29, 2023 from http://www.flux.utah.edu/listarchives/creduce-dev/msg00284.html

[38] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[39] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–12.

[40] Mozilla Security. 2008. Lithium: Line-Based Testcase Reducer. Retrieved May 29, 2023 from https://github.com/MozillaSecurity/lithium

[41] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[42] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 294–305. https://doi.org/10.1145/2931037.2931074

[43] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

[44] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. 2022. XDebloat: Towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4501–4520. https://doi.org/10.1109/TSE.2021.3120213

[45] Jia Le Tian, Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yiwen Dong, and Chengnian Sun. 2023. Ad Hoc Syntax-Guided Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023 (ESEC/FSE 2023)*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, New York, NY, USA.

[46] Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Chengnian Sun, and Shing-Chi Cheung. 2023. Revisiting the Evaluation of Deep Learning-Based Compiler Testing. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, Macao, China, August 19-25, 2023*, Edith Elkind (Ed.). ijcai.org.

[47] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical Extraction Techniques for Java. *ACM Trans. Program. Lang. Syst.* 24, 6 (nov 2002), 625–666. https://doi.org/10.1145/586088.586090

[48] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic Delta debugging. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 881–892. https://doi.org/10.1145/3468264.3468625

[49] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 146–158. https://doi.org/10.1145/3575693.3575740

[50] Zhenyang Xu, Yongqiang Tian, Mengxiao Zhang, Gaosen Zhao, Yu Jiang, and Chengnian Sun. 2023. Pushing the Limit of 1-Minimality of Language-Agnostic Program Reduction. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 97 (apr 2023), 29 pages. https://doi.org/10.1145/3586049

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[52] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498

[53] Mengxiao Zhang, Zhenyang Xu, Yongqiang Tian, Yu Jiang, and Chengnian Sun. 2023. PPR: Pairwise Program Reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023 (ESEC/FSE 2023)*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, New York, NY, USA.