On the Caching Schemes to Speed Up Program Reduction

by

Xueyan Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Program reduction is a highly practical, widely demanded technique to help debug language tools, such as compilers, interpreters and debuggers. Given a program P which 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 62.3% of the generated variants in HDD, a state-of-the-art program reducer, 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 thesis 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, *i.e.*, 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 we periodically remove stale cache entries to minimize the memory footprint over time.

Our evaluation on 20 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 62.3%; correspondingly, the runtime performance is notably boosted by 15.6%. With regard to the memory efficiency, all three methods use less memory than the state-of-the-art string-based scheme STR. ZIP and SHA cut down the memory footprint by 73.99% and 99.74%, compared to STR; more importantly, the highly-scalable, domain-specific RCC dominates peer schemes, and outperforms the second-best SHA by 89.0%.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Prof. Chengnian Sun for his sage advice, careful guidance and patient explanation throughout my graduate study. His meticulousness and enthusiasm towards challenging tasks has had a positive impact in my life.

Words cannot express my gratitude to my thesis readers, Prof. Mei Nagappan and Prof. Shane McIntosh, who generously provided feedback and expertise during the development of my project.

Many thanks to colleagues in Pluverse group for editing support and valuable feedback. I am also grateful to SWAG group for moral support.

Table of Contents

List of Tables

List of Figures

Chapter 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 [43, 31, 39].

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 [43, 31, 37, 6, 35, 39]. 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, 28]. This reason is that a bug-triggering test program in C needs to have only thirty lines of code on average [38]; whereas in practice, such a test program collected from real-world programs or generated by automated compiler testing techniques [2, 42, 26, 7] usually has at least several thousand lines of code. Without program reduction, it is a challenging task for developers to investigate 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 [36], fuzzing [33], program understanding and slicing [3].

Unfortunately, program reduction is computationally expensive and can even take days to finish reducing a program [42, 25]. 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. (More details on the algorithm is discussed in §2.2.) In the above process, the procedure of checking a variant against ψ is referred to as *a query to the property* in this thesis, 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 [31, 17, 18, 39].

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 62.3% and 23.6% of the generated variants in HDD [31] and Perses [39] are duplicates in our benchmark, as 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 of caching scheme for program reduction is string-based caching [19], *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 [34]. A large program P, unfortunately, is rather common; in extreme cases, program reducers crash due to Out-of-Memory Error (OOM). An effective and efficient caching scheme for program reduction is thus necessary.

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 [13, 10]. 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, 23], 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.

Insight 1 Every variant is derived from the current 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.

Insight 2 As a result, when a program becomes 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).

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 considerably 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, the foundation of refreshable caching that further minimizes the memory footprint of caching during program reduction.

We have implemented the proposed caching schemes on top of HDD and Perses, two state-of-the-art program reduction algorithms. Our evaluation on 20 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 62.3% and 23.6% in HDD and Perses respectively. The runtime performance is notably boosted by 15.6% and 13.8%. As for the memory efficiency, caching scheme ZIP (using 3.95 GB on average) and SHA (39.8 MB) cut down the memory footprint by 73.99% and 99.74% in HDD, compared to the baseline STR (15.17 GB). Furthermore, the highly-scalable, domain-specific RCC (4.4 MB) dominates peer schemes, and it outperforms the second-best SHA by remarkably 89.0%. A similar pattern of memory consumption is observed in Perses.

Contributions. This thesis makes the following contributions.

- 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 20 real-world C compiler bugs demonstrate that caching help avoid issuing redundant queries by 62.3% and boost the runtime by 15.6%. Caching schemes ZIP and SHA cut down the memory footprint by 73.99% and 99.74% against the baseline; the domain-specific RCC further outperforms the second-best SHA by 89.0%.

- We have made our implementation, benchmarks, and evaluation scripts publicly available for reproducibility and replicability at `https://github.com/uw-pluverse/perses`

Chapter 2

Preliminaries

2.1 Sequences

This section introduces preliminary knowledge about sequences, since, in the rest of the thesis, 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 $t_i \in \Sigma, 1 \leq i \leq n,$ and $i \in \mathbb{N}$. Notation-wise,

`index`	$p[i]$ denotes t_i, the i-th element in p; i starts from 1.
`size`	$\lvert p \rvert$ denotes the number n of elements in p, which is also referred to as the *size* of p.
`slice`	$p[i : j]$ $(i \leq j \leq \lvert p \rvert + 1)$ 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 $p_1 = \langle t^1_1, t^1_2, \cdots, t^1_m \rangle$ and $p_2 = \langle t^2_1, t^2_2, \cdots, t^2_n \rangle$, $p_1 + p_2$ denotes the concatenation of p_1 and p_2, namely, $p_1 + p_2 = \langle t^1_1, t^1_2, \cdots, t^1_m, t^2_1, t^2_2, \cdots, t^2_n \rangle$.
`equality`	$p_1 = p_2$ if $\lvert p_1 \rvert = \lvert p_2 \rvert \wedge \forall i \in [1, \lvert p_1 \rvert]. \, p_1[i] = p_2[i]$

Definition 2.1.1 (Subsequence)**.** *A sequence $p_1 = \langle t^1_1, t^1_2, \cdots, t^1_m \rangle$ is a* subsequence *of another sequence $p_2 = \langle t^2_1, t^2_2, \cdots, t^2_n \rangle$ if and only if there exists integers $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ where $t^1_1 = t^2_{i_1}$, $t^1_2 = t^2_{i_2}$, $\cdots$, $t^1_m = t^2_{i_m}$. 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 2.1.2 (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 2.1.1 (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 2.1.3 (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. $\langle 1, 2, 3\rangle < \langle 1, 3, 3\rangle$, and $\langle 1, 2\rangle < \langle 1, 2, 3\rangle$

2.2 Program Reduction

In this thesis, 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 denotes 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 $\mathsf{dom}(\tau)$ denotes the domain of τ, and $p \in \mathsf{dom}(\tau)$ implies that the program transformation τ is applicable on the program p.

In this thesis, 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) [43], Hierarchical Delta Debugging (HDD) [31], Generalized Tree Reduction (GTR) [17], Chisel [16], and Perses [39].

One exception is C-Reduce which supports program transformations out of this category [35]: For example, C-Reduce uses Clang [29] 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.

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** $\mathsf{min} \notin \mathsf{dom}(\tau)$ **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** min

2.2.2 Program Reduction without Cache

Algorithm 1 lists the common, overall workflow of program reduction. Most language-agnostic program reduction algorithms [43, 31, 37, 17, 39, 16] 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 what tokens to delete are typically 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 [19], 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). Given that program reduction tools generate a vast number of variant programs and majority of them fail the property test, 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 5.2, STR requires 690 MB memory to cache variant programs in Perses; due to differences in supported program transformations, it requires considerably more memory in HDD, exhausts a memory heap of 44 GB, and triggers an out-of-memory (OOM) error during program reduction.

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 $\mathsf{min} \in \mathbb{P}$ *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 $\mathsf{cache} \leftarrow \varnothing$
4 **while** true **do**
5     $\mathsf{prev} \leftarrow \mathsf{min}$
6   **for** $\tau \in \mathbb{T}$ **do**
7   **if** $\mathsf{min} \notin \mathsf{dom}(\tau)$ **then continue**
8         $p \leftarrow \tau(\mathsf{min})$
9         $cache\_key \leftarrow$ a string which is the source code of p
10   **if** $cache\_key \in \mathsf{cache}$ **then continue** // p has been tested before.
11   **if** $\psi(p)$ **then** $\mathsf{min} \leftarrow p$
12   **else** $\mathsf{cache} = \mathsf{cache} \cup \{cache\_key\}$ // p does not preserve ψ, and is thus cached.
13   **if** $|\mathsf{prev}| = |\mathsf{min}|$ **then return** min

Chapter 3

A Motivating Example

We illustrate how duplicate programs are generated during the program reduction process with an example in Figure 3.1. This example includes one original program in Figure 3.1a, and a property of interest that the program exits with zero returned. Figure 3.1b–3.1j show nine variants sequentially generated in the program reduction process; note that the program reduction process is greatly 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 3.1a, and this program exits with zero.

Step 1–3: Three variants as shown in Figure 3.1b, Figure 3.1c and Figure 3.1d 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 3.1d is generated for the first time, and will be repeatedly generated later.

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 3.1h 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 3.1i 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 example program reduction run, a program as shown in Figure 3.1d 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 thesis.

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 [19].

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 3.1d 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 [13, 10], 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, 23], *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, $\langle 1, 11, 21, 22\rangle$, $\langle 1, 11, 16, 17\rangle$ or $\langle 1, 11, 14, 15\rangle$ throughout the program reduction process.

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 $\langle 1, 11, 21, 22\rangle$, 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 3.1e 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 $= \{p_1, p_3\}$.

3.2 Challenges of Caching Variants during Program Reduction

In practice, caching variants is usually complicated and challenging. Unfortunately, providing 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, 42, 26, 7].

When the initial program P is large, the total number of queries usually increases considerably, and thus the difference in memory footprint between different caching schemes can be amplified. For example, the subject clang-27137 in Table 5.2 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.

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

(a) $p_0, \psi(p_0) = \text{true}$

```c
int main() {
}
```

(b) $p_1, \psi(p_1) = \text{false}$

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

(c) $p_2, \psi(p_2) = \text{false}$

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

(d) $p_3, \psi(p_3) = \text{false}$

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

(e) $p_4, \psi(p_4) = \text{true}$

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

(f) $p_5, \psi(p_5) = \text{false}$

```c
int main() {
   return a + 0;
}
```

(g) $p_6, \psi(p_6) = \text{false}$

```c
int main() {
   int a = 0;
   return 0;
}
```

(h) $p_7, \psi(p_7) = \text{true}$

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

(i) $p_8, \psi(p_8) = \text{false}$

```c
int main() {
   return 0;
}
```

(j) $p_9, \psi(p_9) = \text{true}$

Figure 3.1: An 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.

Chapter 4

Methodologies

This section details the design of the three caching schemes proposed in this thesis, 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) [13, 10]. 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. And §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, 23]. 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.

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.[1]

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.*, $p \sqsubset \mathsf{min}$.

Insight 2 For any program p, *s.t.*, $p \not\sqsubseteq \mathsf{min}$, 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`

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

Table 4.1: 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 $= \langle 1, 6, 21, 22 \rangle$																				
p_2	•	•	•	•	•	•						•	•	•	•	•	•	•	•	•	•
	Encoding $= \langle 1, 7, 12, 22 \rangle$																				
p_3	•	•	•	•	•	•	•	•	•	•											•
	Encoding $= \langle 1, 11, 21, 22 \rangle$																				
p_4	•	•	•	•	•	•	•	•	•	•						•	•	•	•	•	•
	Encoding $= \langle 1, 11, 16, 22 \rangle$																				

(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 $= \langle 1, 6, 16, 17 \rangle$															
p_3 p_5	•	•	•	•	•	•	•	•	•	•						•
	Encoding $= \langle 1, 11, 16, 17 \rangle$															
p_6	•	•	•	•	•						•	•	•	•	•	•
	Encoding $= \langle 1, 6, 11, 17 \rangle$															
p_7	•	•	•	•	•	•	•	•	•	•	•			•	•	•
	Encoding $= \langle 1, 12, 14, 17 \rangle$															

(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 $= \langle 1, 6, 14, 15 \rangle$													
p_3 p_5 p_8	•	•	•	•	•	•	•	•	•	•				•
	Encoding $= \langle 1, 11, 14, 15 \rangle$													
p_9	•	•	•	•	•						•	•	•	•
	Encoding $= \langle 1, 6, 11, 15 \rangle$													

(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 $\langle 1, 7, 12, 22 \rangle$.

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 4.3.2 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

Definition 4.3.1 (Interval-Based Encoding)**.** *Given the minimal program* min *as the base program and a program* p, s.t., $p \sqsubseteq$ 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.

Definition 4.3.2 (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 4.3.3 (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 $e = \langle 1, 4, 6, 9\rangle$, 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 4.3.4 (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., $\not\exists e'.\, expand(e') < expand(e)$*. Note that $expand(e') < expand(e)$ is the lexicographic order defined in definition 2.1.3*

Example. Table 4.1 lists three sets of encodings *w.r.t.* three different base programs, and Table 4.1a shows the compact encoding of four programs *w.r.t.* p_0. We take p_2 as a concrete example to illustrate definition 4.3.1 and definition 4.3.4. In Table 4.1a, the canonical interval-based encoding of p_2 is a compact array $e = \langle 1, 7, 12, 22\rangle$:

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 4.3.4. 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 $e'' = \langle 1, 6, 11, 22 \rangle$, which satisfies the interval-based encoding in definition 4.3.1. 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 thesis. 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 4.1a the encoding of p_3 *w.r.t.* p_0 is $\langle 1, 11, 21, 22 \rangle$, whereas in Table 4.1b its encoding *w.r.t.* p_4 is $\langle 1, 11, 16, 17 \rangle$ and the one *w.r.t.* p_7 is $\langle 1, 11, 14, 15 \rangle$ in Table 4.1c.

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 $\mathsf{min}[\texttt{min\_index}] \neq p[i]$ 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 4.3.1, especially the third condition in the definition, *i.e.*, $\sum_{i=1}^{|e|/2} \mathsf{min}[e[2i-1] : e[2i]] = p$.

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

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 4.3.1 (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* $\mathsf{min}_i \neq \mathsf{min}_j$ *and* $t_i < t_j$*. Then* min_j *is a proper subsequence of* min_i*,* i.e.*,* $\mathsf{min}_j \sqsubset \mathsf{min}_i$*.*

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 $h = \langle \mathsf{min}_i, \mathsf{min}_{i+1}, \mathsf{min}_{i+2}, \cdots, \mathsf{min}_j \rangle$, where $\forall k \in [1, |h|) : h[k+1] \sqsubset h[k]$. Based on the transitivity property in lemma 2.1.1, we can prove $\mathsf{min}_j \sqsubset \mathsf{min}_i$. $\square$

Example. in the contrived program reduction process in Figure 3.1, 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 4.3.2 (Safety of Cache Refresh)**.** *Let* min *denote the minimal program at any time* t *during a program reduction process. If* $p \not\sqsubset \mathsf{min}$*, 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 4.3.1, we know $\mathsf{min}' \sqsubset \mathsf{min}$. Based on the transitivity of the subsequence relation in lemma 2.1.1, we can further conclude $p \sqsubset \mathsf{min}$, which contradicts the condition $p \not\sqsubset \mathsf{min}$ in the theorem. $\square$

Again in Figure 3.1, 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 4.3.2, 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 4.1, p_2 only appears in Table 4.1a *w.r.t.* p_0, but not in the other two tables.

Algorithm 3: Program Reduction with RCC

```
Input: P: the program to be reduced.
Input: ψ : ℙ → 𝔹: the property of interest.
Output: A minimal program min ∈ ℙ s.t. ψ(min)
𝕋: a set of deletion-based program transformations defined in §2.2.1
min ← P
cache ← ∅
while true do
    prev ← min
    for τ ∈ 𝕋 do
        if min ∉ dom(τ) then continue
        p ← τ(min)
        cache_key ← CompactEncode(min, p)
        if cache_key ∈ cache then continue
        if ψ(p) then
            min ← p
            // Refresh cache with the new minimal program.
            cache ← RefreshCache(cache, prev, min)
        else cache ← cache ∪ {cache_key}
    if |prev| = |min| then return min
Function RefreshCache(old_cache, prev, min):
    Input: old_cache: the cache used previously
    Input: prev: the previous min
    Input: min: the current/new min
    cache ← ∅
    for encoding ∈ old_cache do
        p′ ← CompactDecode(prev, encoding)
        if p′ ⊄ min then continue
        cache ← cache ∪ {CompactEncode(min, p′)}
    return cache
```

Algorithm 4: Compact Encoding and Decoding

```
Function CompactEncode(min, p):
    Input: min: a program, s.t., p ⊏ min
    Input: p: a program to compute an encoding for.
    Output: The canonical, compact encoding of p w.r.t. min
    result ← []
    min_index ← 0
    i ← 0
    while i < |p| do
        // scan for the start of the next interval.
        while min[min_index] ≠ p[i] do
            min_index ← min_index + 1
        result ← result + [min_index]
        // scan for the exclusive end of the next interval.
        while min_index ≤ |min| ∧ min[min_index] = p[i] do
            min_index ← min_index + 1
            i ← 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 ← []
    for i ← 1 to |encoding|/2 do
        start ← encoding[2 * i − 1]
        end ← encoding[2 * i]
        p ← p + min[start : end]
    return p
```

Chapter 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 have also conducted ablation experiments to investigate the effect of the two main components of RCC: compact encoding and cache refreshing.

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 [19]. To validate the generality, we implemented the proposed caching schemes (ZIP, SHA, and RCC) on top of DD, HDD and Perses, state-of-the-art program reduction algorithms. However, our evaluations mainly focus on HDD and Perses, as they are effective for reducing structured inputs whereas DD is not [19].

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 memory footprint as the worst-case space complexity, and we also profile the history of memory consumption over time to reveal the trend of memory consumption and the scalability.

RQ2 (Program Reduction Efficiency): *Which caching scheme offers the most speedup in program reduction?*

We run HDD and Perses without caching or with different caching schemes on the benchmark and measure 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): *How do compact encoding and refreshable caching in RCC affect the memory efficiency?*

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 use the benchmark suite collected by Perses [39], which is also used in Chisel [16]. It consists of 20 subjects, each of which is a large C program that triggers a bug in a stable compiler release. These programs have 94,486 tokens on average, and triggers either a crash or miscompilation bug in compilers. The sizes of subjects and the diverse types of bugs make sure that this benchmark is representative of real-world use scenarios of program reductions.

Experiment Environment. All experiments were carried out on a desktop running Ubuntu 20.04 LTS with an AMD Ryzen 5 3600 CPU and 48 GB RAM. The heap size of the JVM was limited to 44 GB.

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

5.2 RQ1: Memory Efficiency

We study the memory efficiency in two aspects. ① we measure the peak memory footprint of different caching schemes to investigate their worst-case space complexity. ② we study the scalability *w.r.t.* to the size of input programs to evaluate which scheme has better scalability when the input size increases.

5.2.1 Memory Footprint

Table 5.1 shows the peak cache size in Kilobytes (*a.k.a.*, the memory consumption/footprint of caching) of different caching schemes in HDD and Perses.

Memory Footprint in HDD. All proposed caching schemes outperform the state-of-the-art STR scheme (shown in Table 5.1). RCC demonstrates the best memory efficiency by reducing the peak memory footprint by 99.97%. On average, the peak cache size of HDD+RCC is minimal, around 4.4 MB (as shown in column 5), compared to HDD+STR, which requires 15.17 GB (in column 2). SHA is the second best alternative scheme, requiring 39.8 MB on average. Although ZIP reduces the memory footprint by 73.99%, it still consumes 3.95 GB on average.

Note that HDD+STR even exhausts the entire heap of 44 GB and triggers OOM on three subjects, clang-27137, gcc-70127 and gcc-70586. We exclude the three subjects when computing the mean, and standard deviation for HDD+STR; thus, the actual statistical numbers of memory footprint in HDD+STR are expected to be much larger. The three proposed caching scheme with different level of compression successfully carry out the program reduction process for all 20 subjects without OOM.

Memory Footprint in Perses. A similar pattern can be observed in the Perses implementation (Table 5.1). With an average cache size of 3.84 MB (in column 9), Perses+RCC outperforms Perses+STR by 97.96% in terms of memory consumption, followed by Perses+SHA (4.3 MB) and Perses+ZIP (35.2 MB).

Table 5.1: Peak Cache Size (KB) in HDD and Perses

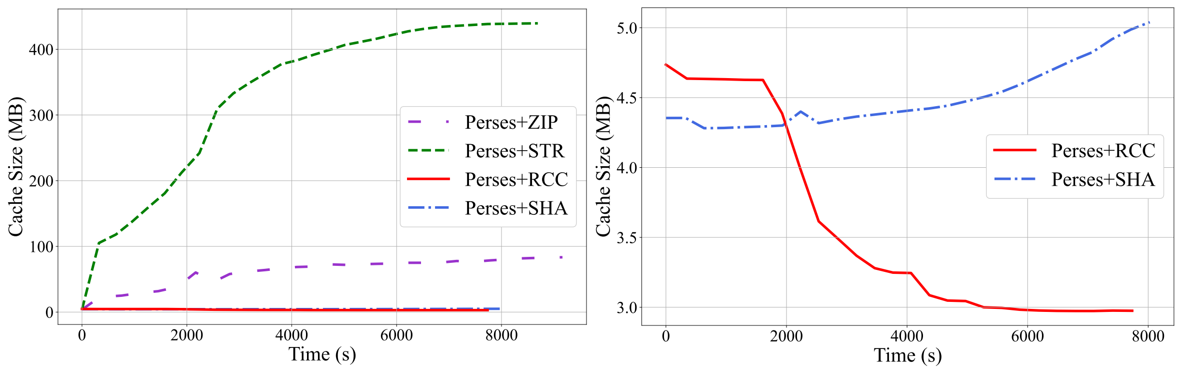
Bug	HDD				Perses			
	STR	ZIP	SHA	RCC	STR	ZIP	SHA	RCC
clang-22382	3,278,569	1,400,121	30,829	3,827	41,926	11,141	3,795	3,206
clang-22704	26,982,067	4,037,096	27,572	4,481	284,370	49,637	4,206	4,516
clang-23309	12,831,908	2,947,573	49,323	5,375	109,749	24,942	4,062	3,347
clang-23353	9,143,238	1,786,036	48,802	3,864	74,954	16,710	3,779	3,279
clang-25900	11,553,691	1,714,629	35,341	4,119	118,617	24,318	4,041	3,667
clang-26350	29,939,211	6,681,769	39,560	4,235	296,302	57,797	4,349	4,028
clang-26760	27,427,226	4,192,774	24,399	4,581	242,213	38,225	4,428	4,717
clang-27137	OOM	18,782,150	85,814	4,441	690,371	128,146	4,933	4,434
clang-27747	1,098,357	244,044	14,459	4,438	28,818	8,917	4,363	4,428
clang-31259	11,920,068	2,025,703	42,355	4,203	90,520	19,864	3,991	3,423
gcc-59903	13,980,675	2,001,552	39,958	4,165	156,417	26,736	4,356	3,498
gcc-60116	8,240,990	1,192,748	40,805	4,549	153,262	28,290	4,467	3,640
gcc-61383	8,927,198	1,488,463	38,577	4,245	83,359	18,278	4,164	3,297
gcc-61917	15,998,852	1,918,573	30,704	4,084	143,610	20,566	4,220	3,721
gcc-64990	39,418,193	3,900,152	36,634	4,339	273,912	32,631	4,395	4,225
gcc-65383	10,018,352	2,046,879	33,274	3,962	76,970	17,604	3,821	3,389
gcc-66186	10,166,856	1,452,940	35,734	4,258	75,912	15,498	4,102	3,418
gcc-66375	16,918,375	3,311,742	44,542	5,035	119,537	34,692	4,212	3,562
gcc-70127	OOM	6,948,302	48,285	4,362	270,622	46,075	4,493	4,276
gcc-70586	OOM	10,837,910	48,842	4,592	439,516	83,440	5,040	4,736
Mean	15,167,284	3,945,558	39,790	4,358	188,548	35,175	4,261	3,840
%Diff. *w.r.t.* STR	0%	73.99%	99.74%	99.97%	0%	81.34%	97.74%	97.94%
Ratio *w.r.t.* RCC	3480.5	905.4	9.1	1.0	49.1	9.2	1.1	1.0

[1] OOM: The statistics of HDD+STR excludes the three subjects with Out-of-Memory Error
[2] %Diff. *w.r.t.* STR : (STR − [Caching Scheme]) ÷ STR × 100%.
[3] Ratio *w.r.t.* RCC : [Caching Scheme] ÷ RCC.

Note that there is a considerable difference in the average memory footprint between Perses+STR (188.5 MB) and HDD+STR (15.2 GB). The reason is that Perses generates much fewer variants than HDD during the program reduction and thus fewer queries [39]. Since the cache size in STR is proportional to the number of generated variants, the cache in Perses with fewer queries is much smaller than that of HDD.

Memory Footprint over Time. Figure 5.1 visualizes the history of memory consumption of caching schemes in Perses over time on gcc-70586 (the subject with the most tokens, *i.e.*, 212,159). It depicts that STR demands a significant, *increasing* amount of memory over 400 MB; ZIP manages to operate under 100 MB, while SHA and RCC require only a fraction of the memory consumption.

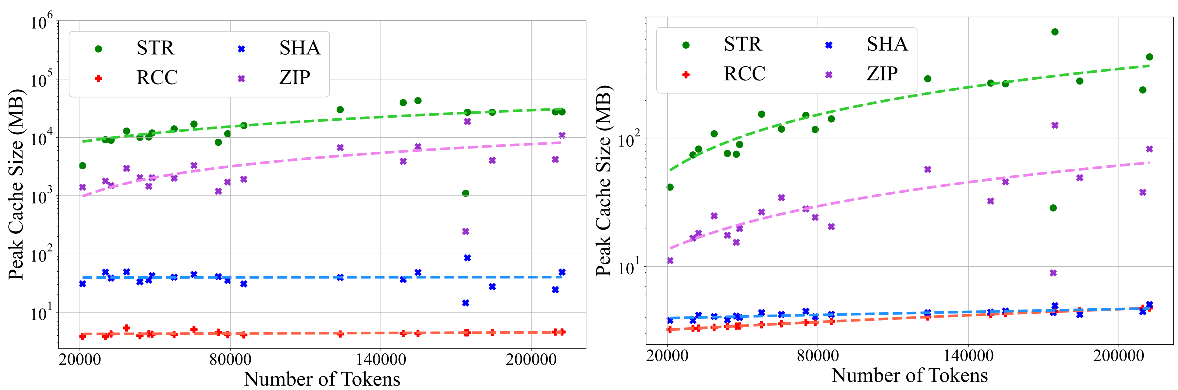
As a program reduction process progresses, RCC continuously uses less memory, whereas other schemes monotonically use more memory over time. The reason is that RCC periodically removes elements from the cache based on Theorem 4.3.2, while peer schemes have no cache refreshing capability.



(a) Overview

(b) Zoom-in details

Figure 5.1: Memory Consumption over Time on subject gcc-70586.



(a) Scalability of Caching in HDD

(b) Scalability of Caching in Perses

Figure 5.2: Peak cache size in log scale (y-axis) v.s. input program size (x-axis) on 20 subjects in HDD and Perses.

5.2.2 Scalability

We study the scalability of different caching schemes by analyzing the correlation between the memory footprint and the size (*i.e.*, number of tokens) of programs.

Figure 5.2a and Figure 5.2b plot the peak cache size of proposed caching schemes and the baseline *w.r.t.* the size of programs in HDD and Perses, respectively. With the information on program sizes in columns 1 and 2 in Table 5.2, we rearranged all the 20 subjects in the ascending order of the number of tokens and plotted the corresponding cache size in log scale. A positive slope indicates a caching scheme requires more memory when the size of input programs increases. STR and ZIP demands more memory when the size of input programs increase; thus, they are more sensitive to the input size changes. SHA with constant 512-bit cache keys has a relative flat curve, but it is subjected to the cache entry accumulation issue as shown in Table 5.1. Lastly, RCC is more scalable; the cache size barely increases when the input program size scales up (*e.g.*, 10 folds, from 21,068 to 212,259 tokens).

Table 5.1 alone also demonstrates that RCC has better scalability than alternative schemes. For instance, for all 20 subjects, HDD+RCC consumes a stable amount of memory between 3.83 MB and 5.03 MB. On the other hand, we observe alternative schemes consume considerably more memory when reducing large subjects. For HDD+STR, the cache size varies from 3 GB to more than 40 GB; for the largest subjects, the experiment script crashes due to memory limitations. We also observe a fluctuation from 244 MB to 18 GB and from 14 MB to 85 MB in HDD+ZIP and HDD+SHA respectively.

Answer to RQ1: While all proposed caching schemes improves the memory efficiency, RCC outperforms alternative schemes, with minimal memory footprint and the best scalability. Compared to STR, it reduces the peak memory footprint by 99.97% and 97.96% in HDD and Perses respectively. Although RCC is effortless to implement, if using existing approaches, SHA provides competitive performance.

Table 5.2: Program Reduction Efficiency Comparison in HDD.

Bug	Original Tokens	HDD								
		No Caching			Caching					
		Query	Time	R_t	Query	STR	ZIP	SHA	RCC	R_t
clang-22382	21,068	277,875	7,447	194	104,365	4,311	4,882	4,391	4,247	194
clang-22704	184,444	187,756	9,454	81	71,332	7,510	8,522	7,553	7,526	81
clang-23309	38,647	425,270	21,869	1,035	170,858	15,416	18,526	15,590	15,284	1,035
clang-23353	30,196	369,429	13,219	143	140,689	8,848	10,309	9,275	8,780	143
clang-25900	78,960	304,086	11,063	462	110,807	7,736	8,854	7,652	7,546	462
clang-26350	123,811	349,363	42,556	429	128,894	38,450	44,248	38,757	38,247	429
clang-26760	209,577	198,665	18,283	303	69,616	16,634	18,331	16,640	16,037	303
clang-27137	174,538	720,875	161,203	531	263,615	OOM	164,175	156,042	151,565	531
clang-27747	173,840	88,391	2,852	332	34,949	1,954	2,133	1,883	1,807	332
clang-31259	48,799	331,105	18,824	590	123,879	13,798	15,284	13,853	13,590	590
gcc-59903	57,581	300,152	13,961	582	115,782	11,078	11,644	11,071	10,596	582
gcc-60116	75,224	301,551	13,876	1,304	118,644	10,155	10,777	10,135	9,968	1,304
gcc-61383	32,449	287,983	12,616	427	111,117	9,406	10,341	9,291	9,048	427
gcc-61917	85,359	235,201	11,735	232	88,062	8,976	10,092	9,126	8,648	232
gcc-64990	148,931	276,496	28,262	410	104,233	26,236	26,987	25,065	24,355	410
gcc-65383	43,942	255,674	10,217	236	96,766	7,069	8,288	7,455	7,158	236
gcc-66186	47,481	271,593	17,055	713	101,520	12,051	13,473	12,668	12,546	713
gcc-66375	65,488	353,516	30,988	856	131,267	23,254	25,636	23,431	22,916	856
gcc-70127	154,816	397,070	64,905	669	143,913	OOM	60,259	57,852	55,910	669
gcc-70586	212,259	374,272	68,442	967	145,284	OOM	69,216	63,133	62,670	967
Mean	100,371	315,316	28,941	525	118,780	13,111	27,099	25,043	24,422	525

[1] All time is measured in seconds. Columns STR, ZIP, SHA, RCC show the reduction time(in seconds).
[2] R_t is the number of tokens after reduction.

5.3 RQ2: Program Reduction Efficiency

We evaluated the program reduction efficiency of the proposed caching schemes by measuring the number of queries, reduction time, and the number of tokens before and after program reduction. Table 5.2 and Table 5.3 show the information on both queries and reduction time of different caching schemes in HDD and Perses. At a glance, the reduced programs have the exact same number tokens with or without caching (comparing column 5 to column 11); it implies the caching is effective and does not change the behavior of reducers.

Table 5.3: Program Reduction Efficiency Comparison in Perses.

Bug	Original Tokens	Perses								
		No Caching			Caching					
		Query	Time	R_t	Query	STR	ZIP	SHA	RCC	R_t
clang-22382	21,068	2,803	459	144	2,315	429	440	427	425	144
clang-22704	184,444	2,276	1,103	78	1,792	1,034	1,060	1,027	1,030	78
clang-23309	38,647	5,967	2,068	473	4,123	1,810	1,860	1,819	1,795	473
clang-23353	30,196	2,754	519	98	2,282	491	503	495	493	98
clang-25900	78,960	2,600	943	248	2,108	898	926	877	898	248
clang-26350	123,811	4,511	4,431	267	3,451	4,135	4,340	4,102	4,131	267
clang-26760	209,577	2,267	1,858	97	1,827	1,814	1,921	1,795	1,788	97
clang-27137	174,538	6,036	9,991	180	4,914	9,456	9,773	9,410	9,349	180
clang-27747	173,840	1,970	710	117	1,555	632	629	630	625	117
clang-31259	48,799	3,214	2,242	406	2,198	1,503	1,511	1,484	1,485	406
gcc-59903	57,581	4,825	3,223	174	3,854	2,991	3,126	2,994	2,975	174
gcc-60116	75,224	6,383	2,753	453	4,410	2,209	2,331	2,229	2,210	453
gcc-61383	32,449	4,338	2,045	497	3,303	1,812	1,854	1,789	1,780	497
gcc-61917	85,359	3,583	1,458	150	2,792	1,417	1,473	1,407	1,369	150
gcc-64990	148,931	3,573	2,219	269	2,649	1,942	2,188	1,990	1,926	269
gcc-65383	43,942	2,658	1,135	143	2,151	1,063	1,079	1,002	1,031	143
gcc-66186	47,481	3,755	2,885	328	2,927	2,037	2,346	2,029	2,013	328
gcc-66375	65,488	4,522	4,174	440	2,918	2,873	2,968	2,875	2,859	440
gcc-70127	154,816	3,106	4,177	301	2,507	3,472	3,583	3,426	3,430	301
gcc-70586	212,259	5,111	6,843	241	4,167	6,062	6,216	6,016	5,992	241
Mean	100,371	3,813	2,762	255	2,912	2,404	2,506	2,391	2,380	255

[1] All time is measured in seconds. Columns STR, ZIP, SHA, RCC show the reduction time(in seconds).

[2] R_t is the number of tokens after reduction.

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 [43, 31, 17, 39]. In this section, we aim to study whether the proposed schemes can effectively avoid the redundant queries in program reductions.

As aforementioned, program reducers like HDD and Perses issue redundant queries. As per numbers in Table 5.2, caching effectively reduces the number of queries issued in HDD by 62.3% from 315,316 (column 3) to 118,780 (column 6). In Perses, caching only issues 2912 (column 6) queries compared to 3813 (column 3) by Perses alone. In conclusion, caching is effective in reducing the number of queries in HDD and Perses.

Notice that STR, ZIP, SHA and RCC issue the same number of queries in both HDD and Perses in Table 5.3; 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.

5.3.2 Reduction Time

Time is another important metrics to measure the efficiency of cache scheme in program reductions. Among three proposed caching schemes and STR scheme, RCC offers the most speedup in runtime performance (shown in Table 5.2 and Table 5.3). Comparing it to program reduction without caching, RCC results in 15.6% faster program reduction in HDD and 13.8% in Perses. Specifically, the average reduction time in HDD without caching is approximately 8 hours (28,941 seconds in column 4, Table 5.2), while RCC shortens the average reduction time to around 6.7 hours (24,422 seconds). Such an improvement in efficiency will facilitate the debugging process and save the time and computation resources for software developers.

Additionally, RCC manages to outperform alternative schemes in terms of reduction time. Besides the three subjects with OOMs, HDD+RCC is 2.1% faster than HDD+STR on available subject data. Using generic compression library, HDD+ZIP and HDD+SHA requires additional 2,677 and 621 seconds on average to complete the program reduction process. We notice that ZIP runtime performance is slower than STR on available subjects due to the lossless compression process. In Perses, the performance of RCC and SHA is comparable, but RCC still surpasses the baseline method STR by a small fraction (1%).

Answer to RQ2: All caching schemes are equally effective in avoiding redundant queries, by 62.3% in HDD and 23.6% in Perses. The domain-specific RCC is the *fastest* and shortens the reduction time by 15.6% in HDD and 13.8% in Perses. The results also confirmed the safety of refreshable cache, *i.e.*, Theorem 4.3.2. Again, SHA is a strong alternative caching scheme, especially for program reducers generating fewer variants than HDD, such as Perses.

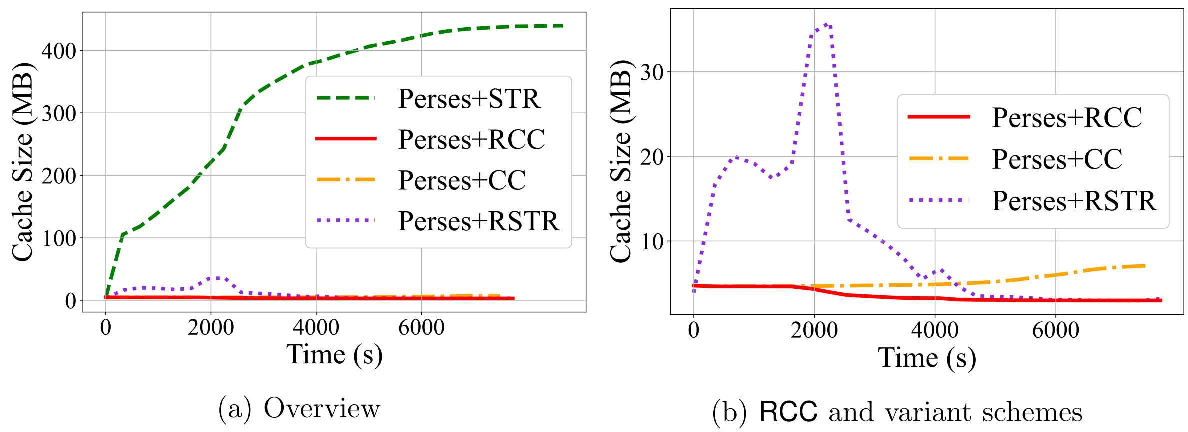


Figure 5.3: Memory Consumption over Time on subject gcc-70586.
Figure 5.3a compares on memory consumption over Time on subject gcc-70586.
Figure 5.3b shows a zoomed-in view by omitting Perses+STR.

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

Table 5.4: Peak Cache Size (KB) of CC and RSTR

Bug	Peak Cache Size in Perses			
	STR	RSTR	CC	RCC
clang-22382	41,926	5,108	3,888	3,206
clang-22704	284,370	51,777	4,993	4,516
clang-23309	109,749	8,390	5,337	3,347
clang-23353	74,954	11,740	3,799	3,279
clang-25900	118,617	14,944	4,112	3,667
clang-26350	296,302	24,543	6,062	4,028
clang-26760	242,213	30,411	5,226	4,717
clang-27137	690,371	43,355	7,882	4,434
clang-27747	28,818	12,172	4,837	4,428
clang-31259	90,520	32,214	4,079	3,423
gcc-59903	156,417	19,913	5,373	3,498
gcc-60116	153,262	40,379	5,676	3,640
gcc-61383	83,359	7,128	4,759	3,297
gcc-61917	143,610	19,628	4,748	3,721
gcc-64990	273,912	53,883	5,503	4,225
gcc-65383	76,970	10,161	3,893	3,389
gcc-66186	75,912	7,942	4,275	3,418
gcc-66375	119,537	9,507	4,730	3,562
gcc-70127	270,622	22,655	5,156	4,276
gcc-70586	439,516	35,831	7,087	4,736
Mean	188,548	23,084	5,071	3,840
%Diff. *w.r.t.* STR	0.00%	90.31%	99.34%	97.94%
Ratio *w.r.t.* RCC	49.1	6.0	1.3	1.0

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

Compact encoding leads to a minimal peak cache size. Figure 5.3a shows the memory consumption of Perses+CC is only a fraction of Perses+STR. Perses+CC considerably reduces the memory footprint (97.3% averagely). However, as shown in Figure 5.3b, without cache refreshing, the memory footprint of Perses+CC accumulates over time and increases from less than 5 MB to 7 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 5.4) 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 5.3a 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 87.8% 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.

Chapter 6

Discussion

6.1 Caching for Delta Debugging

We also studied the impact of caching on DD. But 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 6.1 shows the statistics.

Without caching, DD issues more than five millions queries and takes 21 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 considerably reduced to around seven hours, which is 66.4% faster. Further, RCC outmatches STR in DD in terms of time and memory footprint. Compared to STR (17.7 GB), RCC takes only 3.9 MB. This result further demonstrates that RCC is a general approach for deletion-based program reduction algorithms.

Table 6.1: Comparison of STR and RCC on DD.

	DD	DD+STR	DD+RCC
Query	5,477,887	1,480,695	1,480,695
Time (s)	76,241	26,722	25,648
Cache Size (KB)	N/A	17,692,758	3,883

6.2 Sized-Based Refreshing

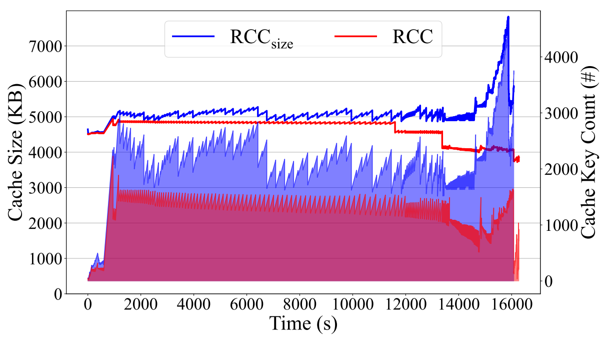
An alternative cache-refresh algorithm for RCC is to record the size of each variant program and remove any programs of which the sizes are equal to or larger than min from cache. This is because the program reduction process starting from min will not generate any variant programs that are larger in 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 $\mathsf{RCC}_{\mathsf{size}}$

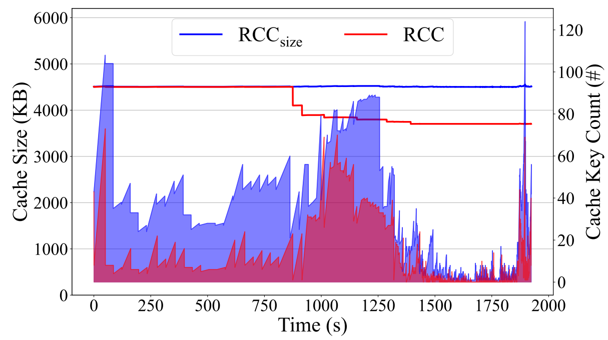
```
1 Function SizeBasedRefreshCache(old_cache, prev, min):
      Input: old_cache: the cache used previously
      Input: prev: the previous min
      Input: min: the current/new min
2     cache ← ∅
3     for encoding ∈ old_cache do
4         p′ ← CompactDecode(prev, encoding)
5         if |p′| ≥ |min| 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 $p' \not\sqsubset \mathsf{min}$. 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 6.1 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.



(a) HDD



(b) Perses

Figure 6.1: Comparison between RCC and RCC$_{size}$ in terms of Memory Footprint (Line) and Cache Key Count (Area) over Time on subject clang-26760.

6.3 Threats to Validity

The subjects used in our benchmark suite may not cover all possible programming languages and bugs for program reduction. To mitigate it, we followed the previous studies in program reduction [35, 24, 39] and used the same benchmark in the previous study in program reduction [39]. The benchmark suite contains 20 C compiler bugs that are collected from real bugs in GCC and LLVM repositories, which consists of programs that are the common size of automatically generated files via fuzzing techniques such as CSmith [42] and EMI [25]. These bugs cover both medium-scale and large-scale compiler bugs. 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.

Chapter 7

Related Work

We survey three lines of the closely related work.

7.1 Caching in Program Reduction

Hodován *et al.* [19] 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 [35].

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 [30, 5, 27], *e.g.*, caching contents in networks for better user experience. In general, an application stores either prefetched data or precomputed 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 [20, 32] 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 program reduction techniques proposed in the literature [18, 24, 17, 39, 43, 4, 41, 40]. Besides the cache, researchers also proposed other methods to improve the performance of program reduction in diverse ways. For example, Hodován *et al.* [19] 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 [21], and recently they further reduced bytecode propositional logic [22].

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.

Chapter 8

Conclusion

This thesis 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 program reduction by compact encoding and refreshable caching. Our evaluation on 20 real-world C compiler bugs demonstrates that caching schemes help avoid issuing redundant queries by 62.3% and boost the runtime performance by 15.6%. For memory efficiency, caching schemes ZIP (using 3.95 GB on average) and SHA (39.8 MB) cut down the memory overhead by 73.99% and 99.74%, compared to the state-of-the-art STR (15.17 GB); furthermore, the highly-scalable, domain-specific RCC (4.4 MB) dominates peer schemes, and outperforms the second-best SHA by 89.0%. As generic, language-agnostic caching schemes, ZIP, SHA and RCC are readily applicable to program reduction techniques and facilitate the program reduction. The implementation of all caching schemes is publicly available at https://github.com/uw-pluverse/perses. Moreover, RCC has been enabled by default in Perses, because of its efficiency and advantageously low memory footprint compared to the others.

References

[1] Dimitris Andreour. Objectexplorer, 2015. https://github.com/DimitrisAndreou/memory-measurer.

[2] Antoine Balestrat. CCG: A random C code generator, 2012. `https://github.com/Merkil/ccg/`, accessed: 2021-01-01.

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

[4] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: in-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.

[5] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stephanie Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *2009 International Conference on Parallel Processing*, pages 462–469. IEEE, 2009.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 197–208, 2013.

[7] Nathan Chong, Alastair Donaldson, Andrei Lascu, and Christopher Lidbury. Many-core compiler fuzzing. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[8] Hong-Tai Chou and David J DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, 1(1-4):311–336, 1986.

[9] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341, 1996.

[10] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

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

[12] GCC. A Guide to Testcase Reduction, 2017. https://gcc.gnu.org//A_guide_to_testcase_reduction, accessed: 2021-01-05.

[13] Mark Adler Greg Roelofs. A massively spiffy yet delicately unobtrusive compression library, Jan 1996.

[14] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.

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

[16] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 380–394, New York, NY, USA, 2018. Association for Computing Machinery.

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

[18] Renáta Hodován, Akos Kiss, and Tibor Gyimóthy. Coarse hierarchical delta debugging. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 194–203. IEEE, 2017.

[19] Renata Hodovan, Akos Kiss, and Tibor Gyimothy. Tree Preprocessing and Test Outcome Caching for Efficient Hierarchical Delta Debugging. *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*, 2017.

[20] Theodore Johnson, Dennis Shasha, et al. 2q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.

[21] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 556–566, 2019.

[22] Christian Gram Kalhauge and Jens Palsberg. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1003–1016, 2021.

[23] J. Kelsey, Shu jen H. Chang, and Ray A. Perlner. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. 2016.

[24] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. Hddr: A recursive variant of the hierarchical delta debugging algorithm. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 16–22, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[26] Vu Le, Chengnian Sun, and Zhendong Su. 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, pages 386–399, New York, NY, USA, 2015. ACM.

[27] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 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*, pages 795–809, 2017.

[28] LLVM. How to submit an LLVM bug report, 2017. https://llvm.org/docs/HowToSubmitABug.html, accessed: 2021-01-01.

[29] LLVM/Clang. Clang documentation – libtooling, 2022. https://clang.llvm.org/docs/LibTooling.html, accessed: 2022-02-22.

[30] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on information theory*, 60(5):2856–2867, 2014.

[31] Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151, New York, NY, USA, 2006. ACM.

[32] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[33] John Regehr. Reducers are Fuzzers – EMBEDDED IN ACADEMIA, 2015. https://blog.regehr.org/archives/1284.

[34] John Regehr. [creduce-dev] cache, 2016. http://www.flux.utah.edu/listarchives/creduce-dev/msg00284.html, accessed: 2021-01-01.

[35] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

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

[37] Mozilla Security. Lithium: Line-based testcase reducer, 2008. https://github.com/MozillaSecurity/lithium, accessed: 2021-01-01.

[38] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, 2016.

[39] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 361–371, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. Xdebloat: Towards automated feature-oriented app debloating. *IEEE Transactions on Software Engineering*, 2021.

[41] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 881–892, 2021.

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[43] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.