# Reproducing the Evaluations for Chisel versus Perses

Funmilayo Olaiya*
Cheriton School of Computer Science
University of Waterloo
folaiya@uwaterloo.ca

Gareema Ranjan
Cheriton School of Computer Science
University of Waterloo
granjan@uwaterloo.ca

## ABSTRACT

In this paper, we investigate **Chisel** and **Perses**, two recent and strong tools for *program debloating* and *program reduction*. We carefully examined if Chisel really was better than Perses as claimed in the research paper that introduced Chisel. While we thought the evaluations the creators of Chisel made of *Chisel versus Perses* were subpar, we revisited this topic to figure out how we might truly reproduce these evaluations *(Chisel versus Perses)* in order to verify the findings again.

We initially examined the Chisel codebase in terms of Reinforcement Learning (RL) and without RL, and then we ran Chisel's benchmarks to analyze how it reduces a given program to see whether there would be a significant performance difference between running Chisel independently or with the help of its RL. On the other hand, Perses was analyzed based on how well it performed on its benchmarks and how extensively it utilised a given program's syntax to guide its reduction process.

We gathered some simple benchmarks *(C programs)* to see how they performed when we reduced them using Chisel and Perses in order to make sure that we did fair evaluations of both tools. We were curious to see how long the two tools took to reduce the programs and how much size reduction they achieved.

Subsequent results show that Chisel did not significantly perform better with its reinforcement learning assistance, and Perses also has the least reduction size when compared to Chisel. Furthermore, Chisel was quicker than Perses both with and without RL.

## KEYWORDS

program reduction, program debloating

## 1 INTRODUCTION

The practice of program reduction is well-known and popular in the software community. It essentially relates to how specific programs can be reduced while retaining some functionality using any given reduction algorithm. Program debloating, in contrast, is nearly identical to program reduction. Still, it gives users the authority and freedom to decide which functionalities they would like to preserve or remove before the program is minimised.

Program reduction is related to the test case simplification process known as Delta Debugging [12], in which the algorithm is fed a failing test case and minimizes the given test case but still retains the failing parts. Moreover, **ddmin**—the Delta Debugging algorithm [12] for simplification of test cases—has been mentioned and used by several modern program reduction tools. Program reduction and program debloating are incredibly versatile concepts with powerful techniques that have given rise to incredible tools like Perses [11] and Chisel [9], which can be used at various stages of software development or testing.

**Perses as a syntax-guided program reduction tool.** The state-of-the-art techniques for program simplification include Delta Debugging (DD) [12] and Hierarchical Delta Debugging (HDD) [10], in which Perses actually makes use of the algorithm that DD introduced.

As Perses employs the syntax of a programming language within a given program to guide its reduction process, Perses also addresses the shortcomings in both DD [12] and HDD [10]. In terms of the size, quality, and speed of the program reduction process, Perses actually performs better than both DD and HDD.

**Chisel as a *custom and debloat* tool.** Programmers can specify the code they want to preserve or remove before the program is minimised using the *custom and debloat* tool called Chisel. The authors of the paper that first introduced the Chisel tool also came up with a set of criteria that a system should conform to, and they assert that Chisel meets these standards. The criteria are *generality*, *naturalness*, *robustness*, *efficiency*, and *minimality*. When Chisel and Perses were compared, the results showed that Chisel was faster in terms of speed and had fewer timeouts on the tested programs.

**Contributions.** This paper makes the following contributions.

- We examined the Chisel codebase [1] to learn more about how it applies reinforcement learning—or whether it actually does—to improve the efficiency of its reduction method. We also investigated how well Chisel's benchmarks [2] fare during the reduction process, and how it performs with and without reinforcement learning.
- Running most of Perses' benchmarks [3] allows us to assess the efficiency of the reduction quality as well as the speed at which these programs are minimised.
- Finally, in order to compare the effectiveness of Chisel and Perses, we compiled a small set of programs to serve as benchmarks [4].

## 2 BACKGROUND AND RELATED WORK

### 2.1 Chisel

Chisel as a program debloating tool was introduced by the authors of the paper titled *Effective Program Debloating via Reinforcement Learning* [9].

The way Chisel works is that it takes in an original program that needs to be minimized and follows a test property. And the output of such an event can be categorised as a minimised version of the

*Conference'17, July 2017, Washington, DC, USA*
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

given original program. Chisel also guarantees that the minimised program must be correct towards the given property.

Figure 1 shows an overview example of how Chisel works, and this figure was gotten from the original paper [9].

One of the main key points of Chisel was to augment program reduction through the use of *Reinforcement Learning* to typically address the shortcomings of other good program reduction tools.

Furthermore, Chisel was tested on ten commonly used UNIX utilities, and each program contains 13-90 KLOC of C source code [9].

In the final evaluations and results of the paper, Chisel outperformed both Perses and C-Reduce mainly in terms of speed and timeouts.

### 2.2 Perses

Perses as a syntax-guided program reduction tool was introduced by the authors of the paper titled *Perses: Syntax-Guided Program Reduction* [11]. Perses typically uses the syntax of a programming language and leverages the grammar to generate only syntactically valid inputs so as to ensure strong program minimisation.

Figure 2 shows an overview example of how Perses works, this figure was gotten from the original paper [11].

Additionally, Perses was tested on 20 large C programs that trigger GCC and Clang bugs and was shown to outperform DD, HDD, and C-Reduce in terms of size and reduction time.

### 2.3 Motivating Example

We illustrate how Chisel and Perses debloat and reduce programs through a simple C program calculator [4]. Figure 3 shows the simple calculator program written in the C programming language. And before we can get the program to run on both Chisel and Perses, we need to create customised test scripts for both.

Figure 4 shows how the Chisel test script for the calculator C program is configured. The *desired* function depicts functionalities of the program that is desired to be kept after minimisation.

Figure 5 shows how Chisel reduces a given program (the calculator program) after the reduction process.

Figure 6 shows how Perses reduces a given program (the calculator program) after the reduction process.

It is clearly noticeable that the reduced sizes of the C program after minimisation by both tools vary differently.

## 3 METHODOLOGY

### 3.1 Approach

*3.1.1 Analysing the Chisel Codebase to understand its Reinforcement Learning System.* As stated in the study that introduced Chisel [9], Chisel uses reinforcement learning to support the reduction process. The authors mentioned using the *Markov Decision Process (MDP)* - an RL framework, for the delta debugging algorithm. The paper's authors didn't really go into great detail about how Reinforcement Learning actually aids Chisel in its reduction process, but they largely discussed Chisel's RL strategy in their paper.

Therefore, the first thing we did was examine the code base to see if RL was actually being used and was being utilised to a maximum extent.

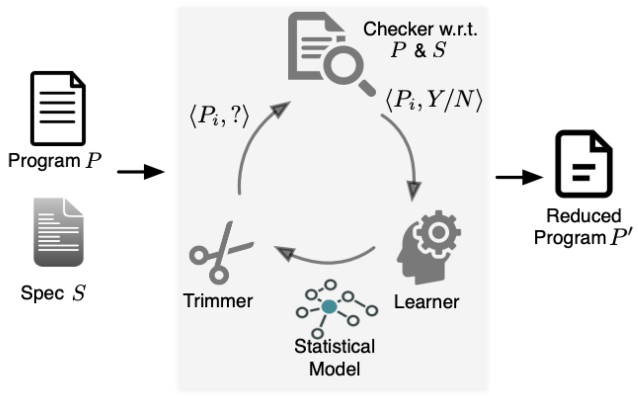


**Figure 1:** Overview of the Chisel system.

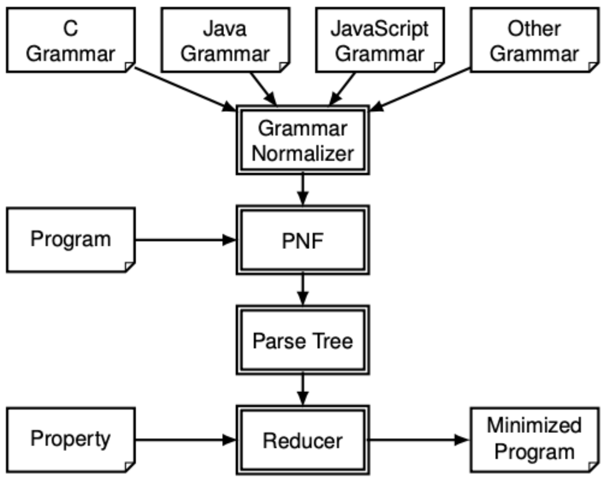


**Figure 2:** Overall Workflow of Perses.

*3.1.2 Exploring and running several benchmarks from both Chisel and Perses.* We ran several benchmarks from both tools only to evaluate their performances first-hand.

**For Chisel**. By following the open documentation for Chisel [5], we attempted to run the given Chisel benchmarks. We had to spend some time figuring this out because the documentation was inadequate. It lacked many of the details we required to explore the codebase, particularly when running benchmarks.

**For Perses**. We also tried using the provided Docker container configurations to run several of Perses' benchmarks, but the Docker setup required a lot of time and space on our machines.

Additionally, following the installation, the setup was not really straightforward because, as we subsequently discovered, certain files on the codebase had not been updated.

Ultimately, we were able to run the Chisel and Perses benchmarks [2][6] by either consulting an expert (for Perses we did this) or modifying some little parts of the source code to make it work on our end (we did this for chisel).

```
int printf(const char *p, ...);
int scanf(const char *restrict, ...);
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *op = argv[2];
    int first = atoi(argv[1]);
    int second = atoi(argv[3]);
    switch (*op)
    {
    case '-':
        printf("%d - %d = %d\n", first, second, first -
             second);
        break;
    case '+':
        printf("%d + %d = %d\n", first, second, first +
             second);
        break;
    case '*':
        printf("%d * %d = %d\n", first, second, first *
             second);
        break;
    case '/':
        printf("%d / %d = %d\n", first, second, first /
             second);
        break;
    default:
        printf("Error! operator is not correct");
    }
    return 0;
}
```

**Figure 3:** A simple C program

```
export SRC=$BENCHMARK_DIR/$BENCHMARK_NAME.c
export ORIGIN_BIN=$BENCHMARK_DIR/$BENCHMARK_NAME
export REDUCED_BIN=$BENCHMARK_DIR/$BENCHMARK_NAME.reduced
export TIMEOUT="-k 0.5 0.5"
export LOG=$BENCHMARK_DIR/log.txt

source $CHISEL_BENCHMARK_HOME/benchmark/test-base.sh

function desired() {
rm -rf out1
clang t.c -o out1
./out1 100 + 100 > temp1.txt
readonly EXIT_CODE="$?"
echo $EXIT_CODE

if [[ "${EXIT_CODE}" == "0" ]] && grep -q "200" temp1.txt ;
     then

return 0
fi

return 1
}

desired
```

**Figure 4:** Sample test script for Chisel

```
int printf(const char *p, ...);
int scanf(const char *restrict, ...);
#include <stdlib.h>

int main(int argc, char *argv[]) {
  char *op = argv[2];
  int first = atoi(argv[1]);
  int second = atoi(argv[3]);
  switch (*op) {

  case '+':
    printf("%d + %d = %d\n", first, second, first + second);
    break;
  }

  return 0;
}
```

**Figure 5:** Chisel's reduced C program

```
int main(int argc, char *argv[])
{
    int first =    argv[1] ;
    int second = atoi(argv[3]);
        printf("%d + %d = %d\n"            , first        );
}
```

**Figure 6:** Perses' reduced C program

*3.1.3 Gathering small benchmarks.* The provided benchmarks by both Chisel and Perses were rather large, and it would have taken a lot of time to customise each of them to run on both tools, so we chose to collect small C programs [4] in order to make a really fair comparison between the two tools.

*3.1.4 Chisel's criteria.* Chisel established five standards for systems to adhere to: *minimality, efficiency, robustness, naturalness, and generality.* We looked at these parameters to see if we could really compare Perses and Chisel through them.

## 3.2 The Gathered Benchmarks

We gathered a few small C programs to serve as benchmarks, and this was done to make it simple for us to accurately assess how Chisel and Perses fair in comparison.

We used random C programs we found online, where we had to modify some code and create Chisel and Perses test scripts for all of them [4].

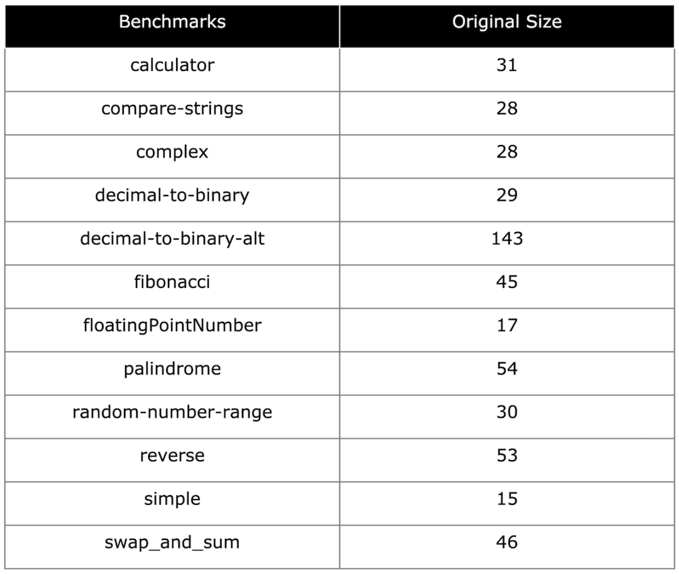
The small benchmark programs and their corresponding sizes are displayed in Figure 7.

## 4 EVALUATION AND RESULTS

### 4.1 Research Questions

We have two research questions that we answer in the following sections.

- How effective is Chisel as compared to Perses in C program reduction?



| Benchmarks | Original Size |
|---|---|
| calculator | 31 |
| compare-strings | 28 |
| complex | 28 |
| decimal-to-binary | 29 |
| decimal-to-binary-alt | 143 |
| fibonacci | 45 |
| floatingPointNumber | 17 |
| palindrome | 54 |
| random-number-range | 30 |
| reverse | 53 |
| simple | 15 |
| swap_and_sum | 46 |

**Figure 7:** The collected benchmarks and their available sizes

```
include <mlpack/core.hpp>
include <mlpack/methods/decision_tree/decision_tree.hpp>
```

```
void ProbabilisticModel::train(int Iteration) {
  if (OptionManager::SkipLearning)
    return;
  Profiler::GetInstance()->beginLearning();
  bool ShouldTrain =
      !(Iteration > 100 && Iteration % (Iteration / 100 +
           1) != 0);
  if ((!OptionManager::SkipDelayLearning && ShouldTrain) ||
      OptionManager::SkipDelayLearning) {
    MyDecisionTree.Train(TrainingSet, TrainingLabels, 2, 1);
  }
  Profiler::GetInstance()->endLearning();
}
```

**Figure 8:** Selected code snippets from the Chisel codebase

- How much does reinforcement learning help Chisel in program reduction?

## 4.2 Chisel's Reinforcement Learning

We tried to evaluate the employed RL algorithm Chisel might have used in the Chisel codebase. And we found some very interesting pieces of code. In the code snippets shown in Figure 8.

**Mlpack** [7] is a package dependency they imported and mostly made use of to possibly help train the dataset and create decision trees. Additionally, it was noted on one of the Mlpack documentation web pages [8] that RL does not fully support Mlpack yet, which begs the question of what kind of RL algorithm the authors implemented.

## 4.3 Reduction Time

Running our collected benchmarks on both systems is a must if we want to accurately compare the speed at which Chisel and Perses can reduce the given programs. The authors of the paper that first introduced Chisel stated that Chisel exceeded Perses in terms of speed and timeout. We also evaluated how Chisel fared in comparison to Perses without employing its RL abilities.

We ran the benchmarks 3 times to collect the average mean of the time used in reduction. Figure 9, Figure 10 and Figure 11 show the tables depicting the data showing the time it took for all use-cases to reduce the benchmarks.

There was no significant time difference in both Chisel with RL or without RL and the reduction time for both was quite similar as shown in Figure 12. If a bar is not shown on the graph, it means that the program timed out and we can not showcase the time.

We confirmed that Chisel is actually much faster than Perses based on the time data we got from the reduction process. This further confirms the authors' claim that Chisel is actually faster than Perses as originally shown in the main paper [9]. Figure 13 depicts the original comparisons of Chisel and Perses in terms of reduction time.

## 4.4 Reduction Size

We also compared the reduction sizes of all use cases; Chisel (with and without RL) and Perses. Figure 14 depicts that Perses outperforms Chisel in actually minimising the benchmarks. It was also very surprising to see that there was no great difference in the reduction sizes of both Chisel (with and without RL) use cases.

## 4.5 Criteria Comparisons

We tried to evaluate if we can compare both tools (Chisel and Perses) using Chisel's set of criteria introduced in the paper [9]. We discovered that we could only evaluate Chisel and Perses in terms of efficiency - which includes *Speed, timeout, reduction size and quality.*

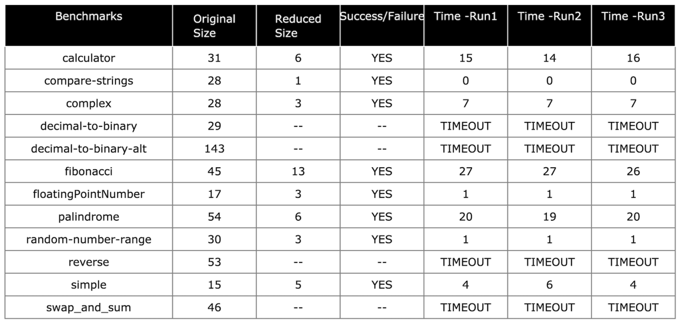
We explain in more detail why we couldn't compare both tools on the other criteria. the biggest reason is that most of these criteria are subjective.

**On Minimality**. Chisel is a **custom and debloat** tool, and it totally depends on what the programmers specify for the reduction process. Meanwhile, Perses takes in no specifications, so to compare both of these tools equally on the bases of Minimality seems unfair.

**On Robustness**. What this condition meant for Chisel is that the newly minimised and created program does not expose new vulnerabilities to a system. We couldn't judge both Chisel and Perses on this because the level at which they reduce the program was absolutely different.

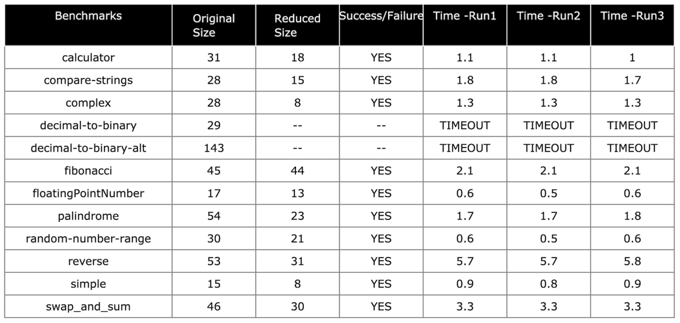
**On Naturalness**. This is also a very subjective criterion. Chisel requires you to specify the code you want to keep before it is minimised, whereas Perses does not require any specifications. So for a minimised program done using Chisel specifically, it absolutely relies on your specifications.

**On Generality**. Both tools handle various programs and specifications, therefore they both perform well in this area, and the degree of performance is entirely dependent on who is rating both tools.



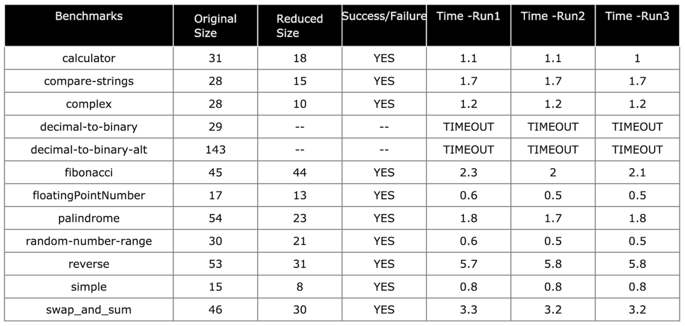
| Benchmarks | Original Size | Reduced Size | Success/Failure | Time -Run1 | Time -Run2 | Time -Run3 |
|---|---|---|---|---|---|---|
| calculator | 31 | 6 | YES | 15 | 14 | 16 |
| compare-strings | 28 | 1 | YES | 0 | 0 | 0 |
| complex | 28 | 3 | YES | 7 | 7 | 7 |
| decimal-to-binary | 29 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| decimal-to-binary-alt | 143 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| fibonacci | 45 | 13 | YES | 27 | 27 | 26 |
| floatingPointNumber | 17 | 3 | YES | 1 | 1 | 1 |
| palindrome | 54 | 6 | YES | 20 | 19 | 20 |
| random-number-range | 30 | 3 | YES | 1 | 1 | 1 |
| reverse | 53 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| simple | 15 | 5 | YES | 4 | 6 | 4 |
| swap_and_sum | 46 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |

**Figure 9:** The time it took Perses to reduce each benchmark three times.



| Benchmarks | Original Size | Reduced Size | Success/Failure | Time -Run1 | Time -Run2 | Time -Run3 |
|---|---|---|---|---|---|---|
| calculator | 31 | 18 | YES | 1.1 | 1.1 | 1 |
| compare-strings | 28 | 15 | YES | 1.8 | 1.8 | 1.7 |
| complex | 28 | 8 | YES | 1.3 | 1.3 | 1.3 |
| decimal-to-binary | 29 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| decimal-to-binary-alt | 143 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| fibonacci | 45 | 44 | YES | 2.1 | 2.1 | 2.1 |
| floatingPointNumber | 17 | 13 | YES | 0.6 | 0.5 | 0.6 |
| palindrome | 54 | 23 | YES | 1.7 | 1.7 | 1.8 |
| random-number-range | 30 | 21 | YES | 0.6 | 0.5 | 0.6 |
| reverse | 53 | 31 | YES | 5.7 | 5.7 | 5.8 |
| simple | 15 | 8 | YES | 0.9 | 0.8 | 0.9 |
| swap_and_sum | 46 | 30 | YES | 3.3 | 3.3 | 3.3 |

**Figure 10:** The time it took Chisel (without RL) to reduce each benchmark three times.



| Benchmarks | Original Size | Reduced Size | Success/Failure | Time -Run1 | Time -Run2 | Time -Run3 |
|---|---|---|---|---|---|---|
| calculator | 31 | 18 | YES | 1.1 | 1.1 | 1 |
| compare-strings | 28 | 15 | YES | 1.7 | 1.7 | 1.7 |
| complex | 28 | 10 | YES | 1.2 | 1.2 | 1.2 |
| decimal-to-binary | 29 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| decimal-to-binary-alt | 143 | -- | -- | TIMEOUT | TIMEOUT | TIMEOUT |
| fibonacci | 45 | 44 | YES | 2.3 | 2 | 2.1 |
| floatingPointNumber | 17 | 13 | YES | 0.6 | 0.5 | 0.5 |
| palindrome | 54 | 23 | YES | 1.8 | 1.7 | 1.8 |
| random-number-range | 30 | 21 | YES | 0.6 | 0.5 | 0.5 |
| reverse | 53 | 31 | YES | 5.7 | 5.8 | 5.8 |
| simple | 15 | 8 | YES | 0.8 | 0.8 | 0.8 |
| swap_and_sum | 46 | 30 | YES | 3.3 | 3.2 | 3.2 |

**Figure 11:** The time it took Chisel (with RL) to reduce each benchmark three times.

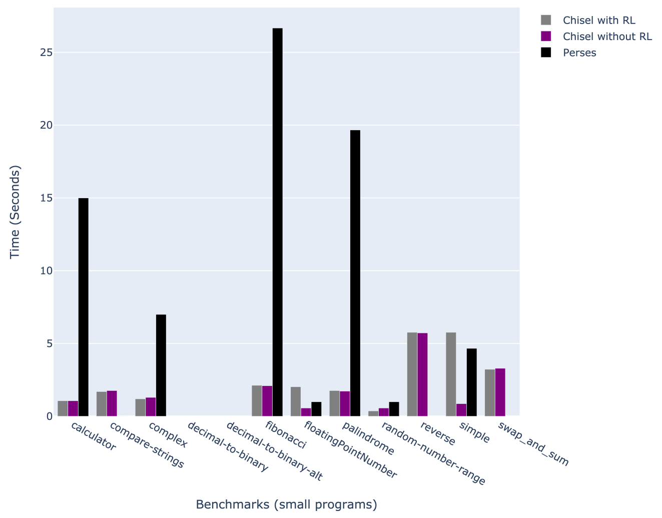


**Figure 12:** Chisel reduces programs faster than Perses using our gathered benchmarks.

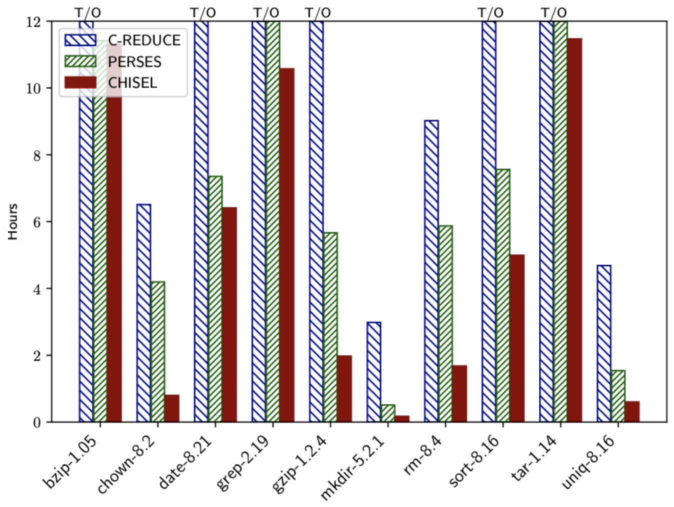


**Figure 13:** Original chart showing that Chisel is actually faster than Perses [9].

## 5 CONCLUSION

In this study, our major objective was to evaluate the effectiveness of two strong tools—Chisel and Perses in how they reduce programs.

Chisel was undoubtedly faster than Perses in terms of reduction time, but Perses was superior in terms of reduced quality and size.

Another intriguing discovery was that even if Perses times out on a particular original program, a minimised program can still be generated, giving Perses a nice advantage.

For future work, we can test if RL with Chisel can have a more profound impact on the results, a possible reevaluation of both Chisel and Perses, but on larger programs.

We think Chisel and Perses are extremely good tools that can be used to debloat or reduce programs, and in order to compare them, a very inclusive, fair, and customised criteria need to be done.

## 6 ACKNOWLEDGMENTS

We thank Professor Chengnian Sun for always being available to answer our various questions, as well as some of our other coursemates who offered suggestions and help on how we may approach these evaluations more effectively.

## REFERENCES

[1] URL: https://github.com/aspire-project/chisel.
[2] URL: https://github.com/aspire-project/chisel-bench.
[3] URL: https://github.com/uw-pluverse/perses.
[4] URL: https://github.com/GareemaRanjan/chisel-perses-comparison.
[5] URL: https://chisel.cis.upenn.edu/.
[6] URL: https://github.com/uw-pluverse/perses/tree/master/benchmark.
[7] URL: https://www.mlpack.org/.

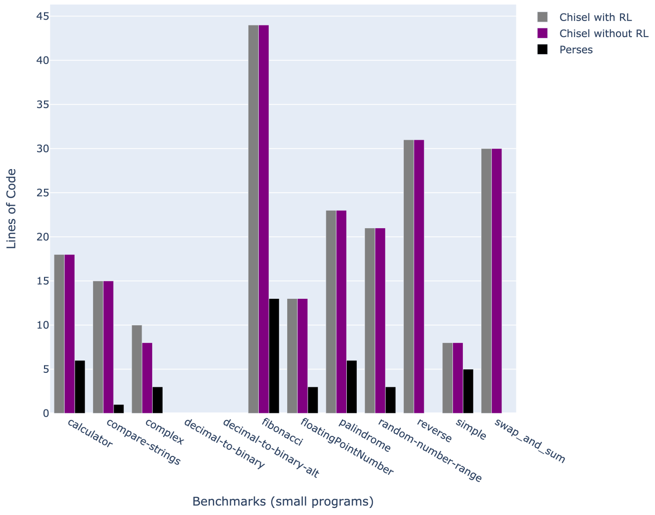


**Figure 14:** Perses reduces the sizes of the benchmarks better than Chisel.

[8] URL: https://www.mlpack.org/gsocblog/deep-reinforcement-learning-methods-summary.html.

[9] Kihong Heo et al. "Effective Program Debloating via Reinforcement Learning". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 380–394. ISBN: 9781450356930. DOI: 10.1145/3243734.3243838. URL: https://doi.org/10.1145/3243734.3243838.

[10] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 142–151. ISBN: 1595933751. DOI: 10.1145/1134285.1134307. URL: https://doi.org/10.1145/1134285.1134307.

[11] Chengnian Sun et al. "Perses: Syntax-Guided Program Reduction". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 361–371. DOI: 10.1145/3180155.3180236.

[12] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input". In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200. DOI: 10.1109/32.988498.