Evaluating the *egg* Equality Saturation Superoptimizer

Luca Hagemans
Supervisor(s): Dennis Sprokholt, Soham Chakraborty
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Superoptimization is the idea of creating the most optimal program possible from a given input program. Equality saturation is a method of superoptimization using rewrite rules and e-graphs. A rewrite rule defines a piece of code that can be rewritten as another part while keeping equal behavior. This is added to an e-graph, a data structure which is able to express a lot of equal programs efficiently using e-nodes and e-classes. *Egg* is a general equality saturation superoptimizer developed in Rust, which has been the basis of multiple extensions. This research first aimed to validate claims made by the creators of *egg* which stated that rebuilding improved equality saturation. Secondly, the research aimed to evaluate the performance of *egg* when supplying a different amount of rewrite rules. The research found that *egg* indeed does improve equality saturation, with an 88× speedup for a part of the algorithm as claimed by the creators, but a claim of 21× overall speedup is an overestimate, as an overall speedup of between 16.4× and 17.4× was achieved. The second part shows that providing *egg* with extra unused rewrite rules slows the algorithm down, although the slowdown seems to decrease when the size of the program increases. The second part also tried to draw a conclusion on adding rewrite rules which will be applied indirectly via multiple other rewrite rules, called implied rewrite rules. Two tests indicated a speedup, but another indicated a slowdown. This means it is unknown if adding implied rewrite rules is beneficial, as no general conclusion could be drawn. Therefore, *egg* seems to benefit from removing unused rewrite rules, and might benefit from adding implied rewrite rules.

1 Introduction

Superoptimization [1] is an idea that exhaustively searches the set of all possible programs to create the most optimal program that has the same functionality as a given input program. This idea is different to the optimizations that traditional compilers apply when compiling programs. Compilers often apply optimizations one by one until the program is optimized [2]. Applying rewrite rules in the wrong order could lead to code which is not optimized to its fullest potential, known as the *phase ordering problem.* Besides this, [3] states: "Developing a bug-free compiler is difficult; modern optimizing compilers are among the most complex software systems humans build". Bugs in compilers could lead to possible optimizations not being applied correctly. Superoptimization aims to address these issues by creating the most optimal program possible for every given input program.

1.1 Equality Saturation

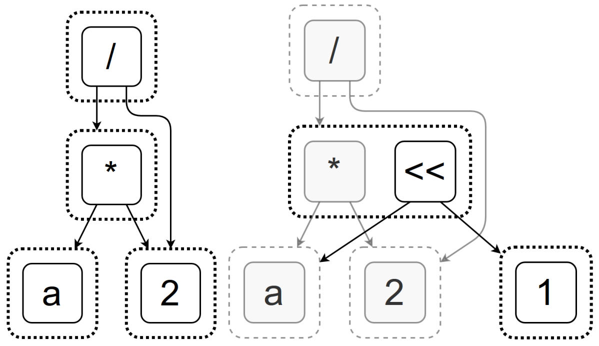
Equality Saturation superoptimizers are a type of superoptimization that use rewrite rules non-destructively using e-graphs to extract the most optimal program possible from a given input program [2]. This type of superoptimizer will be the focus of this research. In this section, the terms defining equality saturation will be explained, concluding with a summary of how equality saturation uses these terms to optimize programs.

Rewrite rules. Rewrite rules are a relation between two program expressions, where the left part can be rewritten as the right part, while still achieving the same result. This is often written as: $\forall x, y, z:$ `lhs => rhs`, where `lhs` is the left hand side, and `rhs` is the right hand side, and x, y, z are variables that can be used in both `lhs` and `rhs`.

An example of a rewrite rule is $\forall a: \ a \cdot 2 \Rightarrow a << 1$ for all a, which means that multiplying the variable a by 2 is the same as bit-shifting a once to the left. These expressions both achieve the same result, but one might be more efficient than the other.

E-graphs. E-graphs are used to express a program by using e-nodes and e-classes. An e-class can contain multiple e-nodes that achieve the same result. An e-node is a part of a program which might have e-classes as children as inputs to that part of the program.

An example e-graph is visible in Figure 1. In Figure 1a, the original e-graph of the program $(a \cdot 2)/2$ is shown, where the boxes with continuous lines are e-nodes and the dotted boxes are e-classes. In Figure 1b, an e-node is added to the same e-class as the multiplication e-node, according to the rewrite rule $\forall a: \ a \cdot 2 => a << 1$.



(a) Original e-graph (b) A rewrite rule is applied

Fig. 1: **Taken verbatim from** [4]. E-graph of program $(a \cdot 2)/2$. From the original e-graph, the rewrite rule $a \cdot 2 => a << 1$ is added to the e-graph

Non-destructive. Non-destructiveness means that the input program is never altered. This is because the e-graph only adds information, and never deletes anything. This ensures the input program is always contained in the e-graph, and makes equality saturation non-destructive. As seen in the example in Figure 1b, the original program is still within the e-graph.

Saturation. Saturation is achieved when there are no more rewrite rules that can be applied to the e-graph that create new information. When the e-graph is saturated, the e-graph contains all possible programs equal to the input program using a particular set of rewrite rules.

Using the terms defined above, the idea of equality saturation can be explained. Firstly an e-graph of the original program is constructed, by creating an e-node within a new e-class for every part of the original program. Next, an algorithm traverses this e-graph to check for e-nodes that can be rewritten using a rewrite rule. If the node can be rewritten, the resulting node of the rewrite rule is added to the same e-class, but only if that node does not already exist in that e-class already. This algorithm runs until saturation, or until timeout, to avoid having the optimizer run infinitely. Lastly, from the resulting e-graph the most optimal program must be extracted. This can be done using a simple greedy extraction, but to guarantee the extraction of the most optimal program Integer Linear Programming (ILP) is often used. ILP maps the problem to a set of binary choices over all e-nodes. It also uses a cost function to determine the optimality of an e-node, where better e-nodes have a lower cost. In this way, the goal is to minimize the sum of the cost of all chosen nodes, using mathematically defined constraints. Using ILP, it is ensured the most optimal program is extracted [5, Sec. 5]. As Equality Saturation is non-destructive, and applies all rewrite rules simultaneously in the e-graph, it avoids the *Phase Ordering Problem*, and is able to provide the most optimal program every time.

1.2 The *egg* superoptimizer

In this project, *egg* will be evaluated. *Egg* is an open source[1] general e-graph implementation in Rust. The *egg* implementation is also capable of equality saturation superoptimization. It is introduced and explained in [4], which iterates on the original concept of equality saturation proposed in [2]. The egg implementation has been used as a basis to improve or create some programs. Three examples of test programs are already given as case studies in the original article. Another program is Diospyros [6], which uses equality saturation to vectorize digital signal processing code automatically. Lastly, Tensat [5] rewrites graphs in deep learning to optimize them. It uses equality saturation to improve on state-of-the-art rewrite methods by optimizing up to 50× faster while creating more efficient programs.

The main improvement of *egg* over the original equality saturation concept is how it restores invariants. The invariants are statements which must hold to assure operations on the e-graph will perform correctly. An e-graph must hold to two invariants, the congruence and hashcons invariants. The congruence invariant dictates that two e-nodes which achieve the same result must be contained within the same e-class. The hashcons invariant states that all canonical e-nodes in the hashcons map must map to the id of their e-class, which allows for congruent e-nodes in the same e-class to be found quickly. Operating on an e-graph can break these invariants, so they must be checked and restored after operating on the e-graph.

The way *egg* improves the invariant restoring is by using *rebuilding* instead of upward merging. Rebuilding is also referred to as rebuilding once per iteration whereas upward merging is also referred to as rebuilding every merge. The technique of upward merging restores the invariants after every operation that could break the invariants. *Rebuilding* instead restores the invariants after an iteration of finding and applying rewrite rules. One iteration will first find all rewrite possibilities, then apply the rewrite rules to the found rewrite possibilities without restoring invariants after each rewrite rule that is applied. Only after all rewrite rules have been applied, the invariants are restored before the next iteration. *Rebuilding* reduces the amount of times the invariants are restored compared to upward merging, and is also able to avoid some cases where upward merging would restore the invariant of the same e-class multiple times.

1.3 Research Question

As *egg* helps improve or is a basis for multiple programs that are mentioned in Section 1.2, understanding the strengths and weaknesses of *egg* could lead to a further improvement of these programs. Therefore, the research question is:

Is the egg *superoptimizer quicker when provided with more rewrite rules?*

To answer this research question, some sub-questions need to be answered first. These questions are:

1. Can the improvement claims from [4] be reproduced?
2. Does *egg* speed up or slow down when provided with extra unused rewrite rules?
3. Does *egg* speed up or slow down when provided with inferable rewrite rules?

We expect that the improvement claims of [4] can be reproduced. Besides that, it is expected that *egg* slows down when provided with extra unused rewrite rules. For the last

[1] https://github.com/egraphs-good/egg

sub-question, it is expected that *egg* speeds up when provided with inferable rewrite rules. Based on the expectations for the sub-questions, the hypothesis of the main question is that *egg* only speeds up when provided with inferable and used rewrite rules, but not when provided with rewrite rules that are unused.

This paper will answer the research question via the different sections. First, in Section 2 the method of answering the research question and its sub-questions are detailed. Next, in Section 3, the results of reproducing the results are given and analyzed. In the following section, Section 4, the results of providing *egg* with different sets of rewrite rules are presented, answering the research question. Then, in Section 5, the responsibility of this research is discussed. In Section 6 the results are discussed and limitations are pointed out. Lastly, Section 7 discusses the main conclusions of the research, and provides some questions for possible future work.

2 Methodology

The first sub-question has been answered using the research artifact of the article [7]. This artifact provides a virtual machine, which is installed with all requirements to reproduce the results of the original paper. In this virtual machine, a script runs the correct tests and logs the results, which can later be interpreted by another provided script. The second script uses the logs to create values further explained in Section 3 and also plots the results in the same plots as used in [4, p.11]. These plots and values will then be compared with the original results.

The results will be reproduced on a 2019 HP ZBook Studio G5 with an Intel Core i7-8750H CPU @ 2.20GHz, with 16 GB of RAM, running Windows 10 Home version 21H2. The virtual machine of the artifact in run in the Oracle VM VirtualBox version 6.1.34 r150636 (Qt5.6.2).

The other two sub-questions have been answered by defining some test programs using the `prop.rs` testing language of *egg*. This testing language implements a small propositional logic language, which supports the AND ($\wedge$) operator, the OR ($\vee$) operator, the NOT ($\neg$) operator, and the implication ($\rightarrow$) operator. Besides these operators, the language supports `true` and `false` as boolean values, and it supports any other symbols as variables. An example of a program written in this language is $(\neg p \wedge q) \rightarrow (p \vee r)$. This language also has a set of rewrite rules that can be applied, such as $\forall p,\ p => \neg(\neg p)$ and $\forall p, q,\ p \vee q => q \vee p$.

Testing programs were written in the `prop.rs` language, which were given to *egg* with a set of rewrite rules to saturate the e-graph. Each program had an e-graph saturated with a selected set of rewrite rules, but also with a set of either all rewrite rules for the `prop.rs` language, called 'the all set', or the set of selected rewrite rules with an implied rewrite rule added, called 'the implied set'. The two different sets of rewrite rules both saturate an e-graph 200 times, while the amount of time taken to saturate the e-graph in milliseconds is stored for each saturation. Using these 200 run-times for both the selected set and either the all set or the implied set, a 95% confidence interval (CI) is calculated. Then, the worst case speedup is calculated by dividing the lower bound of the 95% CI of the all or implied set by the higher bound of the 95% CI of the selected set. This worst case speedup is calculated 5 times for each test, of which the average is taken.

3 Reproduced results

As mentioned in Section 2, [4] claims improvements on the original equality saturation concept in [2]. As explained in Section 1.2, the main improvement of *egg* is the idea of *rebuilding* or rebuilding once per iteration instead of upward merging or rebuilding every merge. This section will compare the claims of improvements made in the article introducing *egg*, by first comparing some values retrieved from reproducing the results, and then comparing some created graphs when reproducing.

3.1 Reproduced values

Reproducing the results claimed in [4] provides four values which evaluate the performance of *egg*. Firstly, values for the congruence and overall speedup of the algorithm are calculated using the run-time on different test cases. The overall speedup indicates how many times faster *egg* is than the original concept of equality saturation, averaged over all test cases. The congruence speedup is the amount of times *egg* is faster in restoring the congruence invariant of the e-graph explained in Section 1.2, also averaged over all test cases. The last two values indicate the Spearman rank correlation coefficient or r_s and the p-value of this correlation coefficient. The correlation coefficient is a number between -1 and 1, where -1 indicates a perfect negative correlation, and 1 indicates a perfect positive correlation. The correlation coefficient describes the correlation between the amount of time spent restoring the congruence invariant and the amount of times the e-graph is rebuilt. The p-value can be used to determine the significance of the correlation coefficient. A positive correlation would mean decreasing the amount of rebuilds will also reduce the amount of time the algorithm spends in restoring the congruence invariant and therefore in total as well. Also, restoring of invariants after each operation could take up the majority of the run time in an existing e-graph implementation [8].

In [4], a 21× overall speedup is claimed, and a 88× congruence speedup is claimed. The speedups achieved for every run along with the averages are visible in Table 1. The 95% confidence interval for the overall and congruence speedup are $(16.432, 17.399)$ and $(87.666, 96.666)$ respectively. Based on these reproduced results, the claim of a 88× congruence speedup lies within the confidence interval, while the claim of a 21× overall speedup does not fall into the confidence interval, and is an over-estimate compared to the reproduced results.

In [4], an r_s of 0.98 is claimed, along with a p-value of $3.6 \cdot 10^{-47}$. Each run of reproducing the results also gave values for these two statistics. The values for each run can be found in Table 1. As $p < .05$ for every r_s gathered in the 10 runs, the correlation coefficient is significant for each of the runs. This means every gathered value of r_s is used in the 95% confidence interval. The confidence interval for r_s is $(0.976, 0.982)$. This means that the claimed value for r_s lies within the confidence interval.

The results given in [4] are mostly valid, as the congruence speedup and the correlation between the amount of times the e-graph is rebuilt and the time spent in restoring the congruence invariant lie within the calculated 95% confidence interval. Only the given overall speedup of *egg* over the original idea of equality saturation is an overestimate based on the calculated 95% confidence interval when running the same tests.

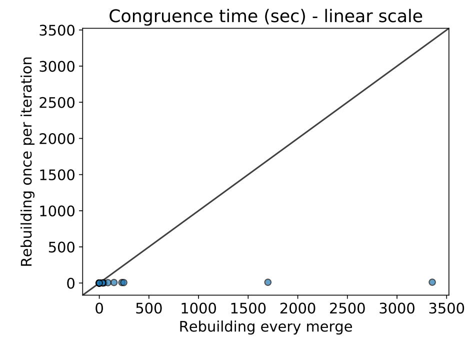
Table 1: Results of reproducing the congruence and total speedup of rebuilding over upward merging, as well as the Spearman correlation r_s and p-value of the relation between the amount of calls to the rebuild function and the time spent in congruence maintenance

Run	Congruence speedup	Total speedup	Spearman r_s	Spearman p-value
1	107.951	18.345	0.970	7.049e-40
2	97.374	17.601	0.979	1.332e-44
3	90.624	17.106	0.981	7.392e-46
4	85.006	16.342	0.982	1.500e-46
5	89.698	16.877	0.972	6.435e-41
6	90.754	17.118	0.981	3.365e-46
7	90.577	16.378	0.982	2.480e-46
8	90.070	16.341	0.983	3.716e-47
9	89.659	16.882	0.979	2.056e-44
10	89.948	16.161	0.978	1.033e-43
Avg.	92.166	16.915	0.979	N/A

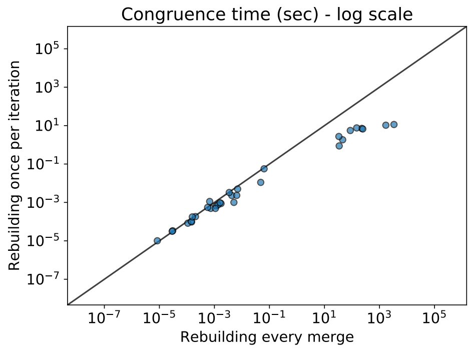
3.2 Reproduced graphs

In addition to the four statistics on the performance of *egg* compared to the original idea of equality saturation, each run created four graphs using the data of the run. The first graph plots the congruence time for different tested programs using rebuilding once per iteration against rebuilding every merge on a linear scale. Every point is a test, every point under $x = y$ indicates rebuilding is faster, as can be seen in Figure 2a. The second graph is the same graph as before but on a logarithmic scale, which can be seen in Figure 2b. The next graph shows the speedup of rebuilding once per iteration over rebuilding every merge against the amount of rewrite rules applied on a logarithmic scale. Each dot also has a trace which shows the speedup change as more rewrites are applied. Every dot above the $1\times$ line indicates a speedup. Dots which have more rewrite rules applied should result in higher speedups, indicating a non-linear speedup when more rewrites are applied, this is shown in Figure 2c. The last graph plots the time spent in congruence maintenance against the amount of times the repair function is called on a logarithmic scale for both rebuilding once per iteration and rebuilding every merge. The correlation indicates that reducing the number of calls to rebuild will result in a reduction of the congruence time, as can be seen in Figure 2d.

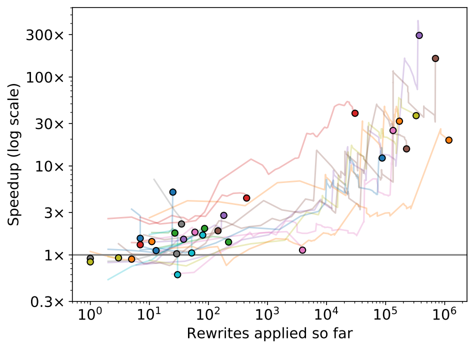
These graphs were also given in [4] and in the virtual machine of [7], which allows for comparison between the given graphs and the newly created graphs. Of the ten runs, the graphs from the run with the least similar results to the original results will be compared. If these graphs indicate the same point as the original graphs, then graphs of the other runs will also convey the same results as they are more similar to the original graphs. In Figure 2, the graphs of the first run are shown to convey the same point as the original graph. These graphs show the same results as the original graphs, just like the graphs of other runs that were even more similar to the original results. A side by side comparison of the original graphs and the reproduced graphs can be found in Appendix A.



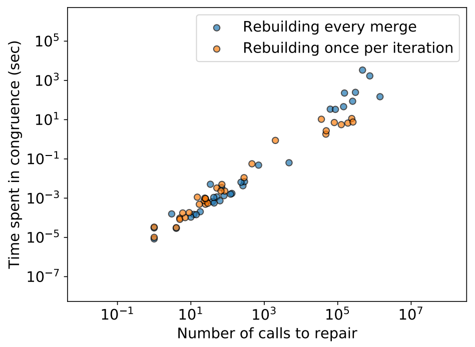
(a) A linear plot of the time taken in congruence maintenance when rebuilding once per iteration/rebuilding against rebuilding every merge/upward merging.



(b) A logarithmic plot of the time taken in congruence maintenance when rebuilding once per iteration/rebuilding against rebuilding every merge/upward merging.



(c) A logarithmic plot of the speedup of rebuilding over upward merging against the amount of rewrites applied. Each point has its the difference in speedup traced over applying new rewrites.



(d) A logarithmic plot of the time spent in congruence maintenance against the number of times the e-graph is rebuilt for both rebuilding once per iteration/rebuilding and rebuilding every merge/upward merging.

Fig. 2: A figure that shows the graphs reproduced in the first run of reproducing the results. The graphs convey the same points as the graphs in the original paper: rebuilding generally provides speedup over upward merging, and this speedup increases non-linearly with the amount of rewrite rules applied.

4 Results of testing *egg*

To test the *egg* superoptimizer, we benchmarked a few programs and evaluated the difference in run time to saturate the e-graphs with a different set of rewrite rules. To test the performance of egg fairly, the tests ensure that the resulting saturated e-graph are the same. Firstly, we tested some programs while providing all possible rewrite rules of that language versus the same program with a selected amount of rewrite rules. Then, we tested if adding inferable rewrite rules would make egg quicker or slower. Inferred rewrite rules are rewrite rules that could be applied through other rewrite rules automatically. An example of such a rewrite rule using propositional logic is the contrapositive of an implication. This rule states that $\forall p, q: \; p \rightarrow q => \neg q \rightarrow \neg p$. However, using the rewrite rules $\forall p, q: \; p \rightarrow q => \neg p \vee q$, $\forall p: \; p => \neg(\neg p)$ and $\forall p, q: \; p \vee q => q \vee p$. All these rules work in both directions, such that the right hand side can also be rewritten as the left hand side. Using these rules, you can prove the contrapositive is equal to the original implication, like this: $x \rightarrow y = \neg x \vee y = y \vee \neg x = \neg(\neg y) \vee \neg x = \neg y \rightarrow \neg x$. This means that the rewrite rule of the contrapositive can be inferred using other rewrite rules. We tested *egg* to evaluate if it would be faster with or without these inferable rewrite rules.

4.1 Different amount of rewrite rules

The idea of testing *egg* by removing rewrite rules that do not change the resulting e-graph, is to see if the programs which build on top of egg might need to look into the rewrite rules they are using, as removing some of these rules might help speedup *egg*'s run-time. This test has been done using the prop test language of *egg*, as it is a small language with a decent amount of rewrite rules. The programs that were tested for the speed of the optimizibility are firstly $\neg(\neg a)$, which required only the rewrite rule $\forall a: \neg(\neg a) => a$. Another tested program was $x \rightarrow y$, to see if *egg* could retrieve the contrapositive $\neg y \rightarrow \neg x$ using the rewrite rule $\forall p, q: \; p \rightarrow q => \neg p \vee q$ and the commutative property of the or operator.

The two programs were tested like mentioned in Section 2, and the worst case average speedup for the first program was 44%. For the second program, a worst case average speedup of 3.84% was achieved. Some extra invalid tests can be found in the repository linked to in Section 2. These tests are not valid as the two different sets of rewrite rules did not create the same e-graph.

4.2 Adding inferred rewrite rules

Adding inferred rewrite rules to the rewrite rules provided to *egg* might lead to a reduction in run-time, as it could lead to less iterations being needed to create the same e-graph as without the inferred rewrite rule.

The first test of implied rewrite rules used the contrapositive example explained above. The contrapositive states that $\forall p, q: \; p \rightarrow q => \neg p \rightarrow \neg q$. Adding this rewrite rule in the set of rewrite rules to saturate an e-graph lead to an average 14.4% worst case speedup.

The second test used De Morgan's laws, which state that $\neg(p \vee q) = \neg p \wedge \neg q$ and $\neg(p \wedge q) = \neg p \vee \neg q$. This test also used the distributive properties of the $\vee$ and $\wedge$ operators that state that $p \vee (q \wedge r) = ((p \vee q) \wedge (p \vee r))$ and $p \wedge (q \vee r) = ((p \wedge q) \vee (p \wedge r))$. Using the De Morgan's laws and the distributive properties, the following relation can be proved: $(\neg a) \vee ((\neg b) \wedge (\neg c)) = \neg(a \wedge (b \vee c))$. This relation is an implied rewrite rule, as this rule can be either given directly or proven via other rewrite rules. Saturating the e-graph with the implied rewrite rule included in the set lead to an average 11.0% worst case slowdown.

The last test tried to prove $((x \rightarrow y) \wedge x) = x \wedge y$. This test uses the rewrite rule $\forall p, q: \; p \rightarrow q => \neg p \vee q$, along with $\forall p: \; p \wedge \neg p => false$ and $\forall p: \; p \vee false => p$. Using these rules together with the distributive properties defined in the explanation of the second test, the implied rewrite rule can be proven. Running *egg* with the implied rewrite rule lead to an average 10.0% worst case speedup over leaving out the implied rewrite rule.

Tests 1 and 3 seem to confirm the hypothesis that adding implied rewrite rules speeds up *egg*. On the other hand, test 2 indicates the opposite of this hypothesis, as *egg* does not benefit from an added implied rewrite rule.

5 Responsible Research

The research should not have any negative impact from an ethical perspective, as there is little interaction with humans, animals or nature, and thus there should be no negative impact on these groups. To reflect on the reproducibility of the research, we have either put all gathered data in the report or at least included all gathered data in the results, and have detailed thoroughly how the research has been done. Because of this, we are confident that any person reading the report should be able to reproduce similar results.

6 Discussion

In this section the results of the tests for this research given in Section 3 and 4 are discussed. In the first part, the results given are interpreted and compared with the hypothesis stated in Section 1.3. The second part discusses some limitations of the research, and aims to explain why some results might differ from the given hypothesis.

6.1 Interpretation of results

In this research, three separate sets of results were created. Firstly, the claimed improvements in [4] of the *egg* superoptimizer over the general equality saturation superoptimizer concepts were reproduced. The only value that did not match up with the claimed results after 10 runs of reproducing was the overall speedup of the *egg* superoptimizer. Although the claim of a 21× speedup could not be reproduced, a overall speedup of between 16.4× and 17.4× is still a large speedup compared to the original idea of superoptimization. This overall speedup along with the large congruence speedup make *egg* a good improvement, and an overall quick superoptimizer.

The second set of results is about providing *egg* with more unused rewrite rules when optimizing different test programs. The results seem to indicate that *egg* is indeed faster when unused rewrite rules are removed from the set of rewrite rules given to the optimization algorithm. When inspecting the two test programs and their results, it seems that the speedup decreases when the size of the program increases. However, due to only two tests being conducted, it is difficult to draw concrete conclusions from these tests.

The last results concern adding inferred rewrite rules to the set of rewrite rules provided to *egg* when optimizing different test programs. As the first and third test give an approximately 14% and 10% speedup respectively, but the second test gives a slowdown of approximately 11%, the results are inconclusive. Besides the difference in speedup, the second test also differed quite in the actual run-times. The second test had an average run time with the selected set of 24768ms opposed to 678ms and 1186ms for the first and third test respectively. This difference in run-time could explain the difference in results, which could mean that the speedup is dependent on the run-time for the specific test. More testing needs to be done to either confirm or deny the hypothesis of the speedup being run-time dependent as well as the original hypothesis given in Section 1.3.

6.2 Limitations of research

The aim of this research was to first reproduce the given results in [4] and then evaluate the strong and weak points of the *egg* superoptimizer. The reproduction of the results resulted in similar numbers for the correlation between the time spent in invariant maintenance and the amount of times the e-graph is rebuilt, as well as for the congruence speedup over congruence maintenance using upward merging instead of rebuilding. For the overall speedup, the claim of a 21× speedup seemed to be an overestimate compared to the results acquired by reproducing. Some limiting factors that can contribute to the difference between the claimed overall speedup and the reproduced overall speedup could be the difference in the hardware that runs the equality saturation algorithm. The original results were produced on a 2020 Macbook Pro with a 2 GHz Intel Core i5 CPU and 16 GB of memory. The reproduced results were produced on a 2019 HP ZBook Studio G5 with a 2.2 GHz Intel Core i7 CPU and 16 GB of memory running Windows 10 Home. The difference in hardware and presumably operating system could explain the difference in the gathered speedup.

Another limitation is the limited amount of runs possible due to time constraints. Each run required a run time of approximately two hours, during which the machine must only run the tests to avoid impacting the results. Time constraints allowed for only 10 runs to be completed, which is a small sample size. As there was some variance in the data that was gathered, more runs would have helped to tighten the bounds for the true possible average.

Lastly, conclusions drawn over different sets of rewrite rules have only been tested on two or three testing programs for using the set of all rules and adding an implied rewrite rule respectively. This is due to time constraints, and the conclusions based on these few tests might not hold when more programs are tested. Especially the claim of less speedup if the program is larger when including all rewrite rules could be very optimistic based on only two tested programs. All of these factors together mean that the current results must be interpreted carefully.

7 Conclusions and Future Work

This research aimed to identify the strong and weak points of the *egg* equality saturation superoptimizer. This was done by firstly reproducing the improvement claims made in [4] which introduces *egg*. The improvement claim of a non-linearly increasing speedup in restoring invariants when increasing the amount of applied rewrite rules was found to be valid. Another claim that was found to be valid was that of a correlation between the time spent in restoring invariants and the amount of times the e-graph was rebuilt. An exception was the overall speedup over the original equality saturation idea which was lower than claimed. Furthermore, the *egg* superoptimizer seems to optimize faster when unused rewrite rules are omitted from the set of rewrite rules provided. How much faster the program is optimized seems to decrease when the size of the program increases. Lastly, adding implied rewrite rules to the provided set of rewrite rules lead to a speedup in some cases, but a slowdown in another case. Therefore, this research is unable to make a general conclusion for including implied rewrite rules when *egg* optimizes programs. In conclusion, the answer to the research question is: *egg* seems to be faster when unused rewrite rules are removed, and it might also benefit from having implied rewrite rules added.

To expand on the current research, more tests can be done on having extra unused rewrite rules for larger programs to evaluate the idea that larger programs lead to a smaller speedup. More tests can also be done on adding implied rewrite rules, as the current tests do not seem to allow for a general conclusion. Like mentioned in Section 6.1, the larger run-time could influence the amount of speedup these test cases got. In general, expanding this research could be done by adding more tests to confirm if the conclusions and hypotheses hold.

Further research on the *egg* could include benchmarking the superoptimizer on different program types such as programs with high cyclomatic complexity, large programs or programs where few rewrite rules can be applied. Another suggestion would be to allow *egg* to optimize widely used programming languages such as C, C++ or Java, and benchmark the performance of the resulting program. Together with this, research can be done on the extraction function, which can evaluate if extraction functions are able to optimize for specific hardware or quickest execution instead of smallest program size.

References

[1] H. Massalin, "Superoptimizer: A look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, 5 Nov. 1987, ISSN: 0163-5964. DOI: `10.1145/36177.36194`.

[2] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," *Logical Methods in Computer Science*, vol. 7, 1 2011, ISSN: 18605974. DOI: `10.2168/LMCS-7(1:10)2011`.

[3] A. Groce, R. van Tonder, G. T. Kalburgi, and C. Le Goues, "Making no-fuss compiler fuzzing effective," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022, Seoul, South Korea: Association for Computing Machinery, 2022, pp. 194–204, ISBN: 9781450391832. DOI: `10.1145/3497776.3517765`.

[4] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, POPL 2021, ISSN: 24751421. DOI: `10.1145/3434304`.

[5] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar, *Equality saturation for tensor graph superoptimization*, 2021. DOI: `10.48550/ARXIV.2101.01332`.

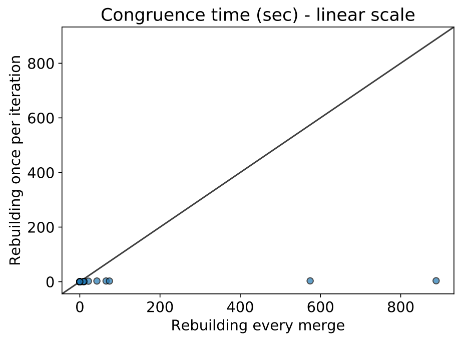
[6] A. Vanhattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, "Vectorization for digital signal processors via equality saturation," 2021. DOI: `10.1145/3445814.3446707`.

[7] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, P. Panchekha, and Z. Tatlock, "Artifact for "Fast and Extensible Equality Saturation"," Oct. 2020. DOI: `10.5281/zenodo.4072013`.

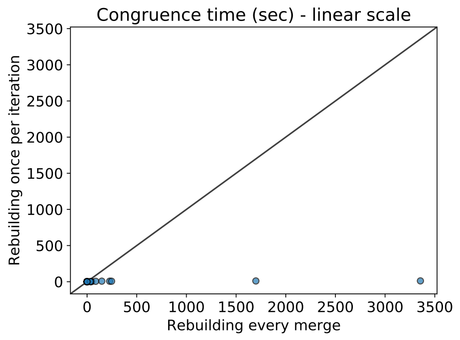
[8] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *SIGPLAN Not.*, vol. 50, no. 6, pp. 1–11, Jun. 2015, ISSN: 0362-1340. DOI: `10.1145/2813885.2737959`.

A Original and reproduced graphs

In this appendix, the original graphs from [4] and the reproduced graphs from the first run are placed. The details of the graphs are discussed in Section 3.2.

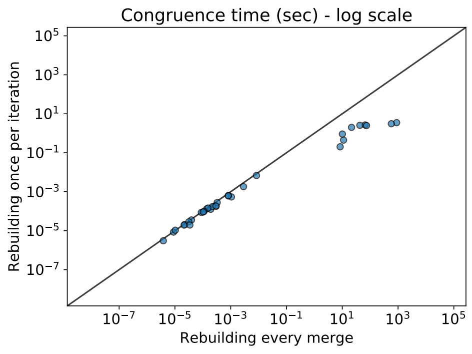


(a) Original graph

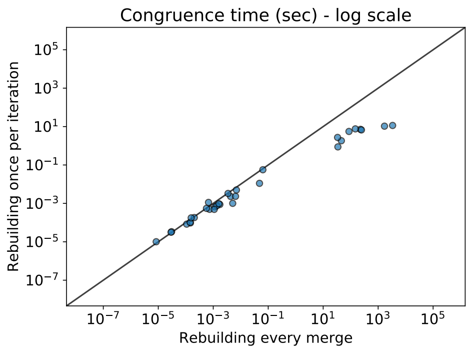


(b) Reproduced graph

Fig. 3: The linear graphs of congruence time.

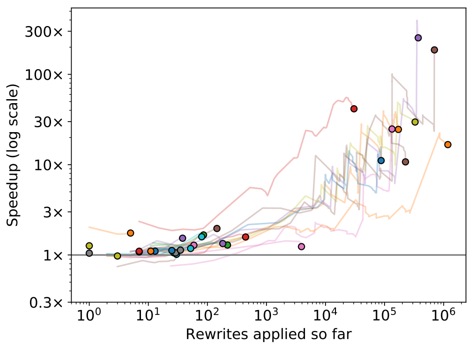


(a) Original graph

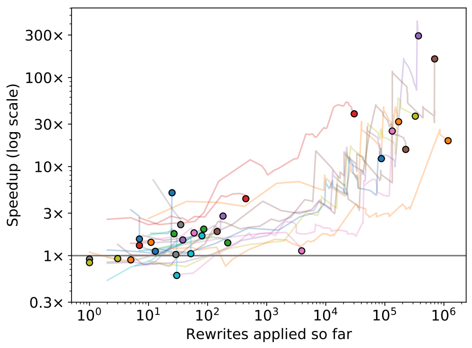


(b) Reproduced graph

Fig. 4: The logarithmic graphs of congruence time.

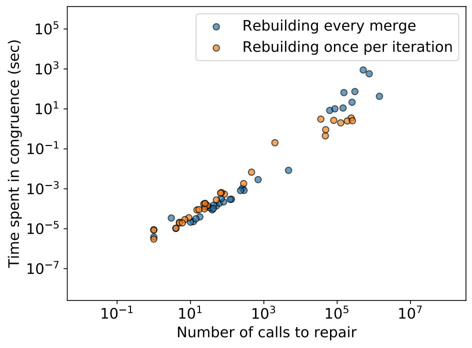


(a) Original graph

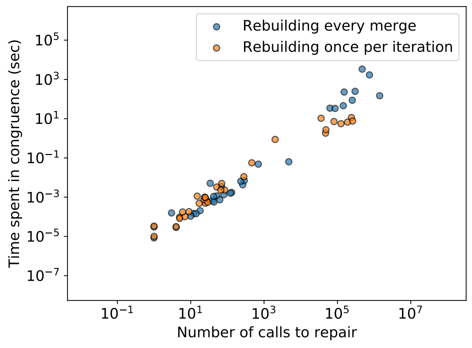


(b) Reproduced graph

Fig. 5: The graphs plotting speedup against the amount of rewrite rules applied, with for each point a traced path of the speedup as more rewrites are applied.



(a) Original graph



(b) Reproduced graph

Fig. 6: The graphs indicating the correlation between the time spent in congruence maintenance and the amount times the e-graph is rebuilt.