Evaluating Stochastic Floating-Point Superoptimization with STOKE

Jop Schaap
Supervisor: Dennis Sprokholt
Professor: 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

The superoptimizer STOKE has previously been shown to be effective at optimizing programs containing floating-point numbers. The STOKE optimizer obtains these results by running a stochastic search over the set of all programs and selecting the best-optimized one. This study aims to find more clearly what floating-point programs STOKE optimizes particularly well and for which ones it fails to find significant rewrites. To answer the research question, STOKE and GCC optimized multiple small programs, and I compared these on execution speed. The results showed numerous cases where STOKE failed to obtain a better optimization than GCC. The results suggest that for specific floating-point functions, there exist limitations in both the test case generator and the STOKE search algorithm that prevent it from finding good optimizations. The findings of this paper suggest further research on the STOKEs test case generator to improve its performance.

1 Introduction

Compilers form one of the primary interfaces that programmers use to develop programs. A compiler's main task is automatically translating a programming language to another one. However, most compilers nowadays include optimizations to speed up the performance of the program they are compiling. Generally, optimizations included in a compiler are restricted to several hand-coded transformations that lead to enhancements in the generated output [1]. An example of such a compiler is the C compiler GCC [2]. Nevertheless, the optimizations compilers apply often do not manage to reach an optimal version because these optimizations are applied independently. Often the best possible code can only be obtained by simultaneously considering multiple optimizations, and it is here where traditional compilers may fail to find the optimum [3, 4].

Superoptimizers do not have this limitation since they enumerate a subset of all programs and select the programs they determine to be equivalent to the original program and the best optimized. The main problem with superoptimizers is scaling them to run on more extensive programs since the search space grows exponentially with the number of instructions. Therefore exhaustive searches are generally unfeasible and instead is opted for alternative approaches [3, 5].

The STOKE [4] optimizer differs from traditional superoptimizers because it uses a stochastic approach to optimize programs. Stochastic approaches have the benefit that they require less time and memory to find a suitable candidate program over that of a traditional optimizer, which in turn allows for bigger programs to be tackled by the optimizer [4].

The focal point of the performed experiments is on the extension in STOKE that adds floating-point support. This extension works by setting a user-defined allowed error. This user-defined allowed error makes STOKE able to optimize floating-point programs in ways that classical compilers cannot [6].

The authors of the extension for floating-point numbers in STOKE have shown that STOKE can generate code that runs faster for some example programs [6]. However, they did not explain why these experience a speedup and if there are programs where STOKE fails to provide a reasonable speedup. Thus, this research aimed to find programs for which STOKE finds good optimizations and when it fails to find such an improvement by comparing it to GCC. Therefore, the research question is:

- What classes of floating-point programs[1] cause STOKE to give well/badly optimized results[2]?

The research question breaks down into the following three sub-questions:

1. How much variation in execution speed do different classes of floating-point programs experience after being optimized with STOKE?

2. How much faster or slower are different classes of floating-point programs after being optimized with STOKE compared to GCC version 12.1?

[1] A class of floating-point programs is a set of programs that heavily rely on certain programming concepts such as; conditional jumps, multiplication, and division or addition and subtraction.

[2] A well optimized result is either a shorter execution speed or a smaller program size, i.e., fewer instructions.

3. How much does an optimized program vary in speed when reducing the floating-point accuracy?

All research questions were answered by performing experiments on multiple floating-point programs and comparing the execution time. For the first sub-question, GCC compiled the C programs hereafter, STOKE optimized them and were they timed. For the second sub-question, were the C programs also compiled with GCC version 12.1 instead of 4.9, which STOKE requires. I answered The final subquestion by optimizing with different levels of floating-point accuracy. The focus in these experiments lies on STOKE-optimized code and code optimized with GCC version 12.1. The optimization quality was measured by timing the execution time of the optimized binary.

The results showed that STOKE failed to find a decent speedup for the programs presented compared to GCC. Whereas the latest version of GCC often found binaries outperforming the results from STOKE. Furthermore, STOKE once failed to provide a binary that replicates the original function and instead got stuck in an infinite loop when given a specific input.

2 Information on STOKE and Floating-Point Numbers

This section provides background information on STOKE and general floating-point arithmetic. First, this section explains the IEEE floating-point representation and unit in the last place (ulp). Secondly, I discuss the internal algorithm of STOKE, and finally, this section describes the test case generation phase of STOKE.

Floating-Point

The IEEE Floating-point representation [7, 8] is the most used representation for real numbers in computers. This representation consists of three parts: the sign, the significant, and the exponent, as seen in [8, eq. (1)]. Because of this representation, are floating-point numbers not evenly spaced across the minimum and maximum range. Instead, they are more present around zero and less at the edges of the number range. The fact that floating-point numbers are not evenly spaced causes operations on these numbers to introduce rounding errors. Due to these rounding errors, do certain properties of regular algebra not hold for floating-point arithmetic, for instance, $a + b = a$ for some a and nonzero b [7, 8].

$$x = -1^{sign} * significant * 2^{exponent} \tag{1}$$

Unit in the Last Place

The rounding errors present in floating-point arithmetic are most often represented in terms of unit in the last place(ulp) [7, 8]. The ulp at x is the difference between the two closest floating-point values of x. For instance, if x would be 1.05 and the two closest floating-point values would be 1.00 and 1.10, then the absolute rounding error would be equal to 0.05, and x would round with an error of 0.5 ulp. The ulp has the advantage over the absolute and relative error in that it has a uniform error representation over the entire range of floating-point values [6–8].

STOKE allows users to specify a value for how much the resulting program is incorrect based on ulp. For instance, if the allowed ulp error is five, the program should always return a value that is at most five ulp different than the original program. This allowed error allows STOKE to optimize floating-point programs in different ways than traditional superoptimizers since the result does not have to be exactly the same as the original [6].

Monte Carlo Markov Chain

STOKE uses a Monte Carlo Markov Chain (MCMC) algorithm known as Metropolis-Hastings (MH) [6]. An MCMC algorithm is an algorithm that allows us to sample from a distribution that we are unable to draw from directly but from which we can evaluate the likelihood of samples from the distribution. The MH algorithm does this by maintaining a current sample, x, and then the algorithm

proposes a random[3] new sample x^* based on x. The suggested x^* is then accepted or rejected based on the target distribution. If x^* is accepted, then x^* becomes the new current sample. Otherwise, a new random x^* is proposed based on x [9].

The STOKE optimizer uses the MH algorithm to find programs that run faster than the original input program. STOKE assigns a cost to each found algorithm based on a cost function and lets the target distribution be the inverse of this cost. This cost function, by default, is the sum of the execution time and a penalty for obtaining an incorrect result based on the distance of the result. STOKE proposes a new x^* by either adding, removing, deleting an instruction, or swapping two instructions. When STOKE finds a new rewrite that is faster than the original and has no penalty for returning incorrect results, then it registers this as the new best and is returned as the optimized result after timing out [4].

STOKE Testcase Generation

As of writing, STOKE contains three fully working methods for generating test cases. The experiments performed used the random generation + backtracking method. This method works by assigning randomly selected values to the input variables and recording the result of the output using the original binary. The random generation + backtracking method runs the fastest and works reliably with code that does not trigger exceptions. Therefore, the experiments use this method as the test-case generator for synthesizing [10].

3 Methodology

The following experiments aim to compare the floating-point optimization capabilities of STOKE to that of the GCC compiler. I performed three experiments with C programs to analyze STOKE and GCC. The experimental setup, including the C programs, was published on GitHub and can be accessed here there[4]. In this section, first, the overall structure of the experiments is explained. Hereafter, this section discusses the four C programs used in the experiments.

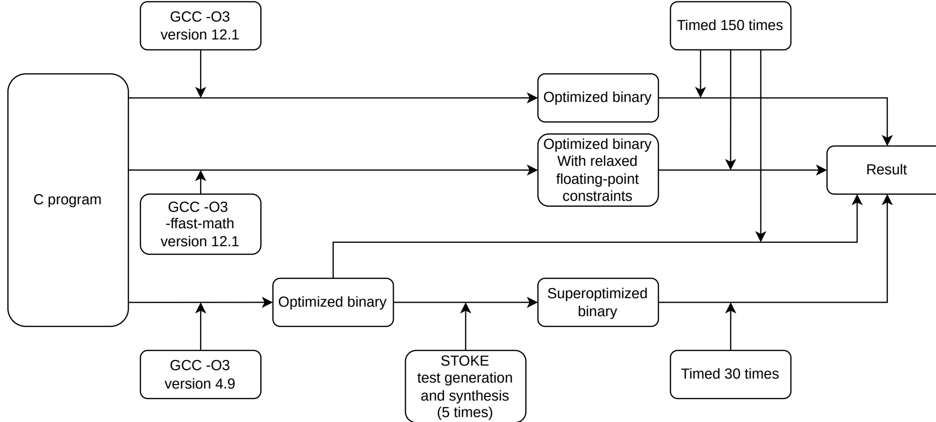
3.1 General Experimental Setup

To compare STOKE to GCC, I wrote multiple C programs. These were then compiled and optimized with STOKE. However, STOKE requires GCC version 4.9 [10], while the latest version as of writing is 12.1. Therefore two different versions of GCC were necessary to compare GCC with STOKE. Thus, I used GCC version 4.9 for STOKE and version 12.1 to compare the speedup, as illustrated in Fig. (1). Furthermore, STOKE ran inside a docker container since it made the execution of STOKE easier. It made the execution of STOKE easier since it prevented the need to set up the required environment, such as the OS and the correct GCC compiler. The time required to execute the program is measured using the clock function defined in the time.h header file [11]. All experiments ran on an Intel I7-8750H operating at 4.100GHz and running Ubuntu 20.04.4 LTS x86_64.

[3] The distribution for this randomness can be chosen at will. However, different distributions will result in faster or slower convergence towards local/global optima.

[4] `https://github.com/JopSchaap/stoke_experimenting`

Figure 1: Visualization of the optimization process. All binaries are optimized and timed 150 times except for the STOKE binary, which is optimized five times and timed 30 times each optimization round.



I estimated the execution speed of the GCC optimized binary in the following procedure. First, I passed the flag -O3 to GCC, and this flag tells GCC to enable all non-standard breaking optimizations [2]. Then, I ran the compiled output files 150 times and timed them using the clock function described above.

Additionally, GCC version 12.1 compiled a new binary with -ffast-math as an additional parameter. This flag allows GCC to relax certain constraints set by the IEEE floating-point standard. I used this flag since STOKE likely would break these constraints if the maximum error value is greater than 0 ulp. Therefore, the speedup obtained using this flag shows how well compilers optimize floating-point programs when relaxing constraints. Finding this speedup is important since STOKE similarly allows for increasing the maximum allowed error, which breaks the IEEE standard on floating-point arithmetic. I finally measured the execution time by measuring the time the function runs using the clock function.

I optimized the code with STOKE by running STOKE in a docker container with GCC version 4.9 installed. First, GCC compiled the original code with the flag -O3. After compilation, STOKE extracted the binary. Hereafter, STOKE generated test cases using the random generation + backtracking method. The function was then optimized and rewritten to the binary file. After optimization, I benchmarked the GCC binary 150 times and the superoptimized binary 30 times. I repeated the test generation, optimization, and running phase, for the STOKE binary five times to prevent the randomness in the MCMC algorithm and the test case generation from impacting the averaged runtime, as illustrated in (1).

Statistical Analysis

I performed a statistical analysis of the obtained results to determine the validity of the results after the experiments. All the statistical analyses were performed using IBM's SPSS software. First, I executed an independent t-test comparing the STOKE binary with the binary produced by GCC version 4.9. I tested the null hypothesis that STOKE had no impact on the execution speed. I rejected the null hypothesis if a statistically significant result was found ($p < 0.05$). Only when the results from STOKE proved to be statistically significant compared to GCC version 4.9 was a t-test performed comparing the STOKE binary to GCC version 12.1 with ffast-math disabled. Hereafter, I executed a t-test evaluating the STOKE binary to the GCC version 12.1 compiled binary with ffast-math enabled. I also performed these steps for the experiment with the maximum tested allowed error. For all performed t-tests, I first ran a Levene's test to determine whether or not the variances can be assumed equal. Hereafter, I either used the t-test that presumes equal variances or the t-test that does not make this assumption.

The independent t-test assumes that the data has a normal distribution. For the experiments, this assumption holds since I timed them over a large number of iterations, or the executed function ran a fixed loop for most of the run time. Therefore, by the central limit theorem [12], the runtimes of the tested functions approximate a normal distribution.

3.2 C Programs

Signum

The first written program was a simple function that takes in a floating-point number and returns the sign of the number, as illustrated in Fig. (2). [5] claimed that non-stochastic super optimizers generated a non-obvious optimization for the signum function that took integers as input. Therefore, the signum function is an intriguing candidate to see the performance of floating-point functions based on integer functions. This signum function is different in that the input is a floating-point number. The intriguing part of the floating-point signum function is that floating-point numbers have interesting edge cases concerning signedness. For instance, in contrast to the integers, do floating-point numbers have signed zeros. Additionally, NaN values exist, which are neither greater than zero, smaller than zero, or equal to zero [7]. It is clear that while looking at the code for the signum function that it should return zero when receiving NaN as input. These edge cases made the signum function an intriguing candidate to show how well STOKE preserves the IEE standard on floating-point arithmetic. I timed the signum function over 10000000 calls since the function itself takes a too short amount of time to execute.

Figure 2: Signum function that takes in a double and returns 1 if it is bigger than zero, -1 if it is smaller and 0 otherwise.

```
int signum(double x) {
  if(x > 0.0) {
    return 1;
  } else if (x < 0.0) {
    return -1;
  } else {
    return 0;
  }
}
```

Range Sum

The second function takes in three values *start*, *end* and *steps*. It performs a sum of evenly spaced values ranging from *start* to *end*, with the *start* being inclusive and the *end* exclusive, as illustrated in Fig. (3). In order to benchmark this function the following input is given: start = 0.0, end = 10e6 and steps = 10e7. I chose the range sum function since it is possible to calculate the result in constant time if we would rewrite the function enough.

Figure 3: Range function that sums up steps values in the range [$start, end$).

```
double range_sum(double start, double end, int steps) {
    double total = 0.0;
    double stepSize = (end - start) / steps;
    for(int i = 0; i < steps; i++) {
        double current = start + ((double)i * stepSize);
        total += current;
    }
    return total;
}
```

The run time of the range sum function scales linearly in time with the *steps* parameter since the loop executes for this amount of steps. However, a speedup exists that would run the calculations in constant time if we perform the arithmetic in the domain of the real numbers. I obtained this faster algorithm by first noting that for summing up integers starting at zero and up to an integer n, there exists a well-known method known as Gauss sum, as seen in [13, eq. (2)].

$$\sum_{i=1}^{n} i = \frac{n * (n+1)}{2} \tag{2}$$

I used the Gauss sum to find a formula that calculates the result of the function in constant time, as seen in eq. (3) and (4).

$$\sum_{i=1}^{steps} (stepsize * i + start) = stepsize * \sum_{i=1}^{steps} (i) + \sum_{j=1}^{start} (j) \tag{3}$$

$$= stepsize * \frac{steps * (steps+1)}{2} + \frac{start * (start+1)}{2} \tag{4}$$

One key observation in this formula is that the result might differ slightly from the function because of floating-point rounding errors [7]. Therefore, GCC without ffast-math and STOKE with no relaxations on the accuracy of the floating-point arithmetic should not be able to find this optimization.

Zero Finder

The third function is the zero finder function which tries to find an x value such that the provided third order polynomial intersects the x-axis, as illustrated in Fig. (4). This function furthermore returns NaN when the provided polynomial is a second or lower-order polynomial since it might not intersect the x-axis in this case. This function is interesting since it contains multiple elements not present in the previous test. These elements comprise returning NaN, having a loop with no clear ending, and containing constants.

The Zero finder function takes a short time to execute, so timing it when running once would likely be unreliable. Hence, similar to signum, I timed the zero finder function over 100000 iterations to make the results more meaningful.

Figure 4: Zero finder function that tries to find a zero output value of a 3th order polynomial.

```
#define CALC(a,b,c,d,x) a * (x * x * x) + b * (x * x) + c * x + d
#define ABS(a) ((a < 0.0) ? -a : a)
double find_a_zero(double a, double b, double c, double d, double epsilon) {
    if (ABS(a) < epsilon) {
        return NAN;
    }
    double current_x = 0.0;
    double current_result = CALC(a,b,c,d,current_x);
    double previous_result;
    double stepSize = 1.0;
    while (1) {
        previous_result = current_result;
        current_x += stepSize;
        current_result = CALC(a,b,c,d,current_x);
        if (ABS(current_result) < epsilon) {
            return current_x;
        }
        if ((previous_result >= 0.0 && current_result < 0.0) || (previous_result <
        ↪  0.0 && current_result >= 0.0)) {
            stepSize = -0.5 * stepSize;
        } else if (ABS(current_result) < ABS(previous_result)) {
            stepSize = 1.5 * stepSize;
        } else {
            stepSize = -1.5 * stepSize;
        }
    }
}
```

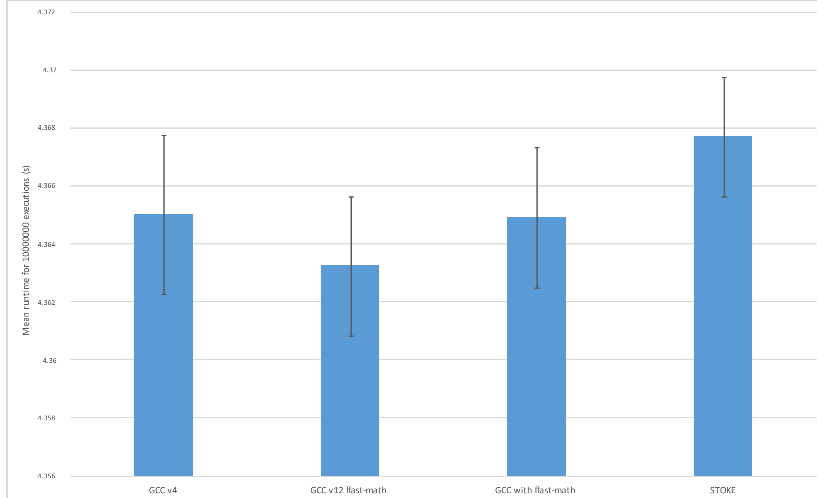
4 STOKE's Performance

The following section will present the results obtained from the experiments. Firstly this section presents the results for signum. Secondly, this section discusses the results of the range sum function, and lastly, it shows the results of the zero finder function. Figures show the found results, and the text discusses the obtained p-values.

Signum

STOKE was able to find an optimization for every iteration of the signum function. STOKE, however, was always unable to generate a function that handled zero's correctly, i.e., one time it returned -1 and the other times $+1$ instead of the expected zero result. The optimized STOKE binaries furthermore took longer to execute than all other binaries, as illustrated in Fig. (5).

Figure 5: Average runtime for 10000000 calls to the signum function after being optimized.



I performed a t-test on STOKE compared to GCC version 4.9 to find the validity of the findings. The results were not statistically significant ($p = 0.123$). Furthermore, the mean difference was -2.670 milliseconds, and the 95% confidence interval has a -6.070 milliseconds lower and 0.730 milliseconds upper bound. Therefore, I cannot reject the null hypothesis that STOKE provides no speedup compared to the binary of GCC version 4.9. This insignificance suggests that the optimizations found had either no or a limited effect on the execution speed.

Range Sum

STOKE was generally unable to find optimizations for the range sum function. One time STOKE was able to find an optimization. However, this optimization caused the resulting program, as illustrated in Fig. (6), to enter an infinite loop. STOKE caused this bug by replacing the loop variable from an integer to a float and replacing the multiplication with addition, as illustrated in line 12 of Fig. (6). The addition performed can, for some inputs, return the same result as the input since floating-point arithmetic does not have the property that $a+b \neq a$ for a non zero b [7]. The automated test generated by STOKE likely often caught this behavior and therefore failed the equality test. However, sometimes it might have passed because of the random nature of the test generation.

Figure 6: The optimization STOKE found(left) and the original compiled version from GCC(right). The STOKE version is one line shorter but has a bug on line 12. STOKE replaced integer addition and multiplication with floating-point addition. This optimization can cause the program to enter an infinite loop on some inputs.

(a) Optimized binary found by STOKE.

```
1  .range_sum:
2    pxor %xmm2, %xmm2
3    subsd %xmm0, %xmm1
4    testl %edi, %edi
5    pxor %xmm3, %xmm3
6    cvtsi2sdl %edi, %xmm2
7    divsd %xmm2, %xmm1
8    jle .L_40064b
9    xorl %eax, %eax
10 .L_400630:
11   cmpl $0x1, %eax
12   addsd %xmm0, %xmm3
13   cmovnzl %edi, %eax
14   pxor %xmm2, %xmm2
15   pcmpeqq %xmm2, %xmm1
16   jne .L_400630
17
18 .L_40064b:
19   movapd %xmm3, %xmm0
20   retq
```

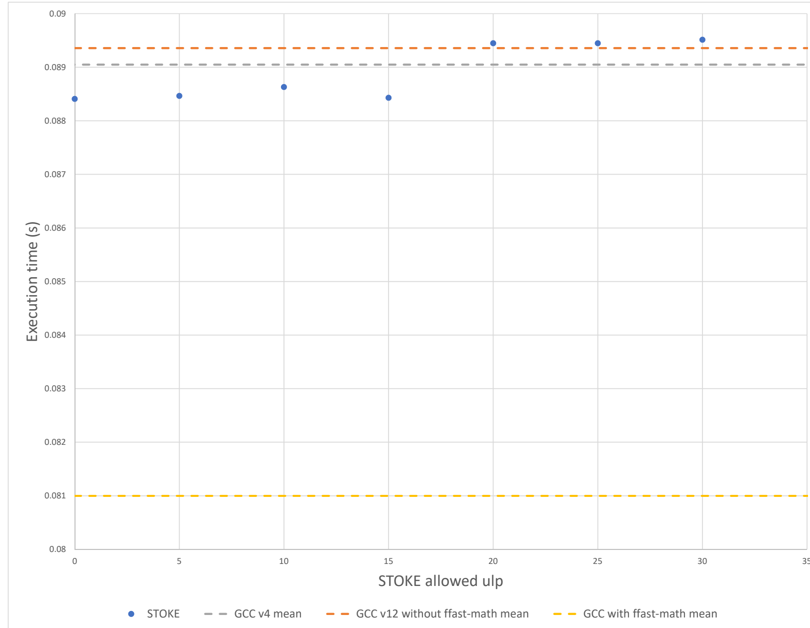
(b) Optimized binary found by GCC.

```
1  .range_sum:
2    pxor %xmm2, %xmm2
3    subsd %xmm0, %xmm1
4    testl %edi, %edi
5    pxor %xmm3, %xmm3
6    cvtsi2sdl %edi, %xmm2
7    divsd %xmm2, %xmm1
8    jle .L_40064b
9    xorl %eax, %eax
10 .L_400630:
11   cvtsi2sdl %eax, %xmm2
12   addl $0x1, %eax
13   cmpl %edi, %eax
14   mulsd %xmm1, %xmm2
15   addsd %xmm0, %xmm2
16   addsd %xmm2, %xmm3
17   jne .L_400630
18 .L_40064b:
19   movapd %xmm3, %xmm0
20   retq
```

Zero Finder

STOKE was able to find optimizations for the zero finder function, which caused speedups compared to GCC, as shown in Fig. (7). When I performed a t-test on STOKE comparing it to GCC version 4.9, I found the results were statistically significant ($p < 0.001$). Additionally, the STOKE binary speed was statistically significant compared to the binary of GCC version 12.1 without ffast-math ($p < 0.001$). Furthermore, the mean difference between the STOKE binary and the GCC binary is: 0.948 milliseconds (a 1.07% decrease), and the 95% confidence interval has a 0.822935 milliseconds lower bound and 1.074108 milliseconds upper bound. However, the STOKE binary was substantially slower than the binary generated by GCC version 12.1 with ffast-math. Furthermore, I found that the STOKE binary with an allowed error of 30 ulp was slower ($p < 0.001$) on average than the binary generated by GCC version 4.9. However, the STOKE binary was not statistically significantly ($p = 0.26$) slower or faster than the binary generated by GCC version 12.1. To summarize, the results show that the STOKE binary received no significant speedup when increasing the allowed error and that the STOKE binary with an allowed error of 0 ulp performed better than both versions of GCC and substantially worse than when ffast-math was enabled.

Figure 7: Averaged runtime of 100000 executions of the STOKE and GCC optimized Zero Finder functions where the allowed ulp for STOKE is altered for multiple experiments (x-axis).



5 Responsible Research

Ethical Aspects

With the ever-increase of software applications in our personal lives, bugs more and more impact users. An example of a software bug that had a significant effect on the users is the bug in Therac-25. Therac-25 was a medical device that contained a software bug that exposed people to extreme radiation levels. The bug in Therac-25 caused multiple people to die and is considered one of the worst bugs in history [14].

Most bugs will not have such a disastrous impact as the bug in Therac-25. However, it is not unthinkable that a bug introduced could have large-scale effects. STOKE has an increased chance of introducing system bugs since it does not formally prove that the original program is equivalent to the resulting binary. STOKE operating on assembly code further increases this risk by complicating manual checks of the correctness of the program because assembly is more complex than higher-level languages. Therefore, users should take special care when programs are optimized using STOKE. The increased chance of introducing bugs in the program shows the importance of studies such as this one that evaluate the capabilities of STOKE and showcase its limitations.

Reproducible

To increase the credibility and reproducibility of the performed experiments, should the data and code required for the experiments be publicly available. Therefore, the scripts for running the experiments are all uploaded to GitHub[5]. Additionally, this repository contains all of the programs discussed previously. It incorporates automation scripts for compiling, running STOKE, and timing the binaries. This repository is open for anyone to use, and I designed it to be expandable for generating data on additional C programs.

[5]`https://github.com/JopSchaap/stoke_experimenting`

6 Discussion

This research aimed to find programs for which STOKE gave helpful and unhelpful optimizations. The results indicate that, for the programs tested, STOKE did not find optimizations that gave meaningful speedups. Furthermore, the study demonstrates that sometimes, STOKE struggles to generate tests that properly find infinite loops in the optimized code. The experiments failed to identify programs that gave significant speedups, but they showed several examples of cases where STOKE fails to generate satisfying results.

The results show that STOKE fails to generate optimizations that provide substantial speedups for the tested functions. The best speedup for the programs under test was for the zero finder function, which only had an execution time reduction of 1.07% compared to the GCC version 12.1 binary. Furthermore, the range sum function has a known optimization that causes the algorithm to execute in constant time instead of polynomial time. However, STOKE did not find this optimization and suggested an optimization that caused the program to loop infinitely. These findings show that STOKE struggled to find optimizations for the given functions and sometimes even produced malfunctioning optimizations.

These results seem to contradict the claims of [6] that STOKE can find significant speedups over GCC. However, the experiments performed in [6] focus on different classes of programs. Namely, the experiments [6] have fewer conditional jumps and generally optimize over loop-free code. Therefore, it is likely that specifically, the class of programs tested in this experiment caused STOKE to produce the unsubstantial optimizations it did. Thus, the results cannot be used to generalize the total capabilities of STOKE and only show the limitations of STOKE for programs similar to the programs in the experiments.

Fig. (7) might suggest that the STOKE binary runtime decreases when the allowed error increases. However, it is more likely that a bias in the measuring setup causes this. The researchers of [15] show that slight variations of aspects of an experimental setup can cause a substantial bias in program run time. Likely, this caused the observed jump in the runtime of the binary for higher values of allowed error because the experiments ran over two days. Therefore it is feasible that the measurement setup had a slight but uncontrollable bias during the second day. However, the speedup found is still valuable since I performed the first 4 STOKE and the GCC measurements on the same day.

The results highlight no cases where STOKE performed exceptionally well compared to GCC. Therefore, the results are less helpful in finding further optimizations for the GCC compiler. However, the converse that STOKE can be improved based on the results is valid. Specifically, the results show an inability of the STOKE test case generator to generate input data for large loops. This inability is likely because STOKE assumes that the program is in an infinite loop after a certain number of loops. Future development could fix this shortcoming in STOKE by generating a couple of test cases that use a larger bound for deciding when the program is in an infinite loop.

Furthermore, the results highlight the inability of STOKE to search programs that contain loops with floating-point numbers. Since even when a simple optimization exists that results in a polynomial speedup, STOKE cannot find it. The authors of STOKE say the following about loops in STOKE: “STOKE identifies loop-free subsequences of the code which it will attempt to optimize” [6, p.7]. This statement suggests that STOKE cannot find an optimization because it does not optimize the loop. However, since finding the constant time optimization for the range sum only involves changing the parts outside of the loop and leaving the loop empty, should STOKE still be able to find it. Further research is needed to establish what prevents STOKE from optimizing these programs and if similar programs exist that will get optimized.

The limited time constrained the methodological choices available for the performed research. Therefore the results should not be interpreted as an assessment of the full capabilities of STOKE. However, the results still are valid for answering the research questions since these aim to find cases where STOKE under or overperforms, which can be seen in the results.

7 Conclusion and Future Research

This research aimed to identify programs for which STOKE optimizes well and when it fails to find a significant optimization. To find these programs were multiple programs written and hereafter optimized using both STOKE and GCC. The results show that STOKE failed multiple times to provide satisfactory results when the function contains a loop. Therefore, I conclude that STOKE generally struggles to optimize floating-point functions containing loops. Furthermore, the results indicate that the STOKE test case generator fails to generate test cases that properly check infinite looping behavior when looping over floating-point numbers.

Additionally, STOKE did not find optimizations that performed better than GCC version 12.1 with ffast-math enabled throughout the experiments. Therefore, the results might suggest that no class of programs exists where STOKE synthesizes exceptionally well. However, STOKE's inability to optimize the given programs does not generalize to all programs since the experiments focused on a small set. Therefore it is possible that for other programs, STOKE matches or even outperforms GCC version 12.1 with ffast-math enabled. Future studies could address this gap in knowledge and focus on different sets of programs to apply STOKE on.

The results indicate that the STOKE test case generator often fails to generate tests for floating-point programs that properly verify the resulting program. For instance, the test case generator sometimes fails to produce tests that check for infinite loops. Therefore, improving the test case generator will likely benefit STOKE since this would cause STOKE to give more accurate results. Furthermore, an enhanced test case generator will allow STOKE to run with a reduced training set which would speed up the search and enables STOKE to check a larger space for optimized programs. Future studies could focus on alternative methods for test case generation to more accurately validate the generated programs and synthesize them with a smaller test set that reduces search time.

References

[1] W. M. Waite and G. Goos, *Compiler Construction*. Texts and Monographs in Computer Science, New York: Springer Science & Business Media, 1996.

[2] "gcc(1) - Linux manual page," May 2022.

[3] S. Gulwani, P. Oleksandr, and S. Rishabh, "Program Synthesis," in *Foundations and Trends® in Programming Languages*, vol. 10 of *Applied Logic Series*, pp. 105–134, Dordrecht: Springer Netherlands, 2017.

[4] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 305–316, Mar. 2013. Number: 1.

[5] H. Massalin, "Superoptimizer: A Look at the Smallest Program," *ACM SIGARCH Computer Architecture News*, vol. 15, pp. 122–126, Oct. 1987.

[6] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Optimization of Floating-Point Programs with Tunable Precision," *ACM SIGPLAN Notices*, vol. 49, pp. 53–64, June 2014. Number: 6.

[7] D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, pp. 5–48, Mar. 1991. Number: 1.

[8] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefévre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Cham: Springer International Publishing, 2018.

[9] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan, "An Introduction to MCMC for Machine Learning," *Machine Learning*, vol. 50, pp. 5–43, Jan. 2003. Number: 1.

[10] E. Schkufza, R. Sharma, B. Churchill, S. Heule, and J. Koenig, "GitHub - StanfordPL/stoke: STOKE: A stochastic superoptimizer and program synthesizer," May 2022.

[11] "time.h(0p) - Linux manual page," May 2022.

[12] S. G. Kwak and J. H. Kim, "Central limit theorem: the cornerstone of modern statistics," *Korean Journal of Anesthesiology*, vol. 70, pp. 144–156, Feb. 2017. Publisher: The Korean Society of Anesthesiologists.

[13] J. DeMaio, "Counting Triangles to Sum Squares," *The College Mathematics Journal*, vol. 43, pp. 297–303, Sept. 2012. Number: 4 Publisher: Taylor & Francis _eprint: https://doi.org/10.4169/college.math.j.43.4.297.

[14] N. Leveson and C. Turner, "An Investigation of the Therac-25 Accidents," *UC Irvine: Donald Bren School of Information and Computer Sciences*, vol. 26, no. 7, pp. 18–41, 1992.

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing Wrong Data Without Doing Anything Obviously Wrong!," *ACM SIGPLAN Notices*, vol. 44, p. 12, Mar. 2009.