Evaluating STOKE

PRZEMYSLAW KOWALEWSKI
Supervisor(s): SOHAM CHAKRABORTY, DENNIS SPROKHOLT
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

`STOKE` is one of the Superoptimizers which are programs that given a function and a set of instructions of a processor, traverse through a space of programs that compute a given function and try to find the optimal usually in terms of execution speed or size of the binary. Authors of `STOKE` make some extraordinary claims. They suggest that it is able to produce programs that are multiple times faster than programs without any optimization, and programs which are at least as efficient as programs produced by `gcc -O3` and sometimes expert handwritten assembly. The goal of this paper is to check these claims. In this paper classes of programs that `STOKE` may handle particularly well and any class of programs that stochastic optimization might not be able to handle will be identified. We conclude from the experiments described in this paper that `STOKE` is able to fulfill that statement in some cases. The searching algorithm of `STOKE` is not always able to find programs that are at least as efficient as programs optimized by `gcc -O3`. `STOKE` works particularly well in programs where a lot consecutive logical and mathematical operations are calculating (e.g. counting bits). It is often not that successful with programs containing loops where it sometimes can't find a solution at all.

1 Introduction

Superoptimizer is a program that given a function and a set of instructions of a processor, traverses through a space of programs that compute a given function and tries to find the shortest one[3]. One of the Superoptimizers - `STOKE`[5] introduces a cost functions and then it uses Markov Chain Monte Carlo[1] sampler (specifically Metropolis-Hastings algorithm) to try to find the solution with the lowest cost value.

MCMC is a sampling algorithm which "draws elements from a probability density function in direct proportion to its value: regions of higher probability are sampled from more often than regions of low probability."[5]. In context of cost function it means that the most frequent samples will be taken from from the areas with lowest value which allows for finding the most optimal solution. When `STOKE` finds a successful rewrite it tries to find a new one based on a function based on the most optimal solution at that time. `STOKE` efficiently traverses the search space thanks to first identifying different regions in it with equivalent programs and then trying to optimize programs within these regions.

This approach is not limited to a single class of programs which is innovation in that field as previous work in that field was usually limited to a performing a search of all programs within some singular class of programs whereas `STOKE`'s authors claim that its approach is universal and can traverse the search space of all possible programs[5]. It saves a significant amount of time on not searching the regions which would not contain correct solutions to the program and thanks to optimizing solutions in different areas of the search space it can find a novel solution to a given problem[5]. Other important constraint of `STOKE` is that the correctness preservation while performance improvement is desired but not required.

In this paper, I will be focusing on evaluating the `STOKE` Superoptimizer. Authors in the original paper claim that it can generate programs a few times faster than the compiler without any optimizations would. They also state that the new program will be at least as efficient as programs produced by `gcc -O3` and in some cases faster than expert handwritten assembly.

[1] MCMC

The goal of this research is to check these claims or have they changed over the years of `STOKE` development and identify classes of programs that it may handle particularly well and any class of programs that stochastic optimization might not be able to handle competently.

This will be done through repeating experiments carried out in the original paper and creating new test cases and testing them to identify the pros and cons of this particular Superoptimizer in context of different program types.

The structure of this paper is as follows, firstly methodology of the experiments will be discussed, then the experimental setup and results will be described. Following that, all the validated experiments from the original paper will be discussed in detail and after that, all the classes of programs that were evaluated will be discussed. In the end, the conclusions of the research will be laid out and potential future work will be discussed.

2 Methodology

There are many factors that make this particular Superoptimizer challenging to properly evaluate. One of them is randomness in traversing the search space, we can make the seed constant for all experiments, which would make them easily reproducible but it could potentially prevent us from finding a more optimal solution. Therefore I have used three different seeds to check for the most optimal solution, that being said it does not fully solve the problem but allows to see if the seed is a big factor in the experiment's result and might require more investigation.

The other factor that influences is the cost function that navigates the search through the search space. `STOKE`'s authors implemented quite a few of them and also allowed combining and assigning weights to them. This means that using one particular cost function may lead to no search results at all, while some other cost functions may lead to finding some attractive results. The most relevant cost functions are:

- correctness - number of bits that differ in the outputs of the target versus the rewrite summed across all test cases,
- latency - an educated guess of the run time of given instruction based on number of processor cycles required to execute given instruction,
- measured - same as latency but measured on given processor,
- size - number of instructions in a rewrite.

The default cost function is defined as "correctness+latency", it was tested in all of the experiments as well as its different permutations such as "correctness+latency+size", "correctness+measured", etc.

There are a plethora of other important factors of which different permutations were tested, such as penalties for wrong results, proposed changes to the program, starting point in the search space (which for all experiments in this paper is a non-optimized program generated by `llvm -O0`), etc. This problem does not affect the evaluation of programs shown in the original publication [5] as the authors provide `STOKE`'s configuration on the project's Github page. However, it does affect the evaluation of different program classes as we never have certainty that the found solution is the pinnacle of the `STOKE`'s possibilities which is an important limitation of this research.

When measuring the performance of both optimized and non-optimized versions of a function, they were ran millions of times such that the execution time of the whole program

is measured in several seconds, then that program is run 10 times and the average of these measurements is calculated to account for any background activity of the operating system that might have slowed down the execution of the program. Following that we calculated 95% confidence interval for these results to account for the measurement inaccuracy (excluding the programs with high computational complexity as they have much higher run-time, in that case duration of one function call was measured).

3 Experimental Setup and Results

All experiments were executed on MSI GS65 Stealth Thin 8RE laptop equipped with Intel Core i7-8750H processor, DDR4 8GB Ram, GTX 1060 with Ubuntu 14.04 with GCC version 4.9 installed. `STOKE`'s source code was retrieved and compiled from the latest commit on June 20, 2022.

To compare the results of the optimization of the functions performed in the original paper [5] configuration files and source code on the previously mentioned GitHub repository was used and then the experiments were ran 10 times each on the computer and measured with `time` command. Average of these measurements was taken and speed-up was computed based on these averages.

To benchmark the functions written to test the `STOKE`'s efficiency with different program classes all functions were ran in a program run which called these functions 10^n times where n is adjusted to make execution time of the program in range from 0.5 second to 10 seconds to mitigate the overhead caused by program loading and starting and all the memory operations. Then that program was run 10 times to account for background activity of the computer which could lead to slow down the execution of the program and the average was taken from these measurements. All the `STOKE` configuration files used to perform the optimizations are published on the TU Delft's GitLab.

Function name	gcc -O3	`STOKE`	Speed-up
p21	2.52	1.23	51%
p23	0.62	0.54	10%-14%
SAXPY	1.30	1.28	−1%-3%
Linked List	3.8	3.79	0%-1%

Figure 1: Program execution time in seconds and speed-up after applying `STOKE`'s optimization when repeating the experiments from the original paper

4 Verification of original experiments

There are multiple experiments carried out in the original `STOKE` paper [5]. In this section results of trying to reproduce them will be presented as well as the authors claim that `STOKE` should produce programs at least as fast `gcc -O3` will be verified. For some experiments the source code and/or assembly code will be provided when those are helpful for explaining the experiments results. If one of the code results is omitted but the reader wants to take a look at them anyway, it can be done by reproducing the experiments which can be done by using the configuration files and code which are hosted on `STOKE`'s GitHub page.

Experiment name	gcc -O3	STOKE	Speed-up
HCC[a]	0.74	0.67	−8%-−13%
HCCDC[b]	5.39	6.13	−13%-−15%
RMA[c]	2.16	2.39	6%-13%
RMA SMALL[d]	1.78	1.86	−3%-−5%
TS[e]	4.98	SEGFAULT	-

[a]High Cyclomatic Complexity
[b]High Cyclomatic Complexity with Dead Code
[c]Random Memory Access
[d]Random Memory Access to a small array
[e]Travelling Salesman Problem

Figure 2: Program execution time in seconds and speed-up after aplying STOKE's optimization when testing different classes

4.1 p21 function

```
int p21(int x, int a, int b, int c){
  return
    ((-(x == c)) & (a ^ c)) ^
    ((-(x == a)) & (b ^ c)) ^ c;
}
```

Figure 3: p21 function

p21 function takes four integer values, performs different logical operations on them and then returns one integer value.

```
# gcc -O3
xorl eax eax
cmpl ecx edi
movl ecx r8d
sete al
xorl esi r8d
negl eax
andl r8d eax
cmpl esi edi
movl edx esi
sete dil
xorl ecx esi
movzbl dil edi
negl edi
movl edi edx
andl esi edx
xorl eax edx
movl edx eax
xorl ecx eax
```

```
# STOKE
xorq rax rax
xchgl ecx eax
xorq rdx rdx
sarq 0x1 rdx
```

Figure 4: p21 function before and after `STOKE`'s optimization

As it can be seen in the figure above on the left we can see that program generated by `gcc -O3` is a pretty naive rewrite of the original function. `STOKE` found a really efficient solution, reducing the number of instructions by 15 and achieving a speed-up of 51% comparing to the original version.

4.2 p23 function

p23 is a function that calculates number of 1 bits in a given number. It is implemented using bit shifts, and operations, additions and subtractions. Assembly code generated by `gcc -O3` will be omitted as it is a naive rewrite of the source code.

```
int p23(int x) {
  int o1 = x >> 1;
  int o2 = o1 & 0x55555555;
  int o3 = x - o2;
  int o4 = o3 & 0x33333333;
  int o5 = o3 >> 2;
  int o6 = o5 & 0x33333333;
  int o7 = o4 + o6;
  int o8 = o7 >> 4;
  int o9 = o8 + o7;
  int o10 = o9 & 0x0f0f0f0f;
  int o11 = o10 >> 8;
  int o12 = o10 + o11;
  int o13 = o12 >> 16;
  int o14 = o12 + o13;
  return o14 & 0x0000003f;
}
```

Figure 5: p23 functions's source code

```
# STOKE
popcntq rdi rax
```

Figure 6: p23 function's assembly after `STOKE`'s optimization

As it can be seen in the figure above `STOKE` manages the most optimal solution which is reduction of all 49 instruction to one - `popctn` which counts number of 1s in the register. Thanks to this a speed-up of at least 12% over `gcc -O3` is achieved which is significant but quite smaller that claimed in the paper which is more than 50%.

4.3 SAXPY function

SAXPY is a linear algebra function which takes a constant a, two pointers to arrays y and x and an offset i and then calculates $x_l = x_l * a + y_l$ where l is an integer from a range of i to $i + 3$.

```
void s(int a, int* x, int* y, int i) {
  x[i+0] = x[i+0] * a + y[i+0];
  x[i+1] = x[i+1] * a + y[i+1];
  x[i+2] = x[i+2] * a + y[i+2];
  x[i+3] = x[i+3] * a + y[i+3];
}
```

Figure 7: SAXPY function

This experiment gave the worst results in comparison to the results in the original paper. The speed-up achieved thanks to STOKE's optimizations was basically non-existent.

4.4 Linked List Traversal

This example in the original paper is meant to show limitations of STOKE. The code of that function given a pointer to the first element, traverses whole linked list and doubles each element in it.

```
while (head != 0) {
  head->val *= 2;
  head = head->next;
}
```

Figure 8: Linked List Traversal

The speed-up achieved from applying that optimization is negligible and in range of statistical error however this was expected due to the nature of this Superoptimizer. STOKE is not able to fully optimize this function but only the most inner fragment of the loop therefore it misses some possible loop optimizations.

5 Evaluating Different Program Classes

The aim of this section is to test how STOKE handles different program classes by writing different programs, optimizing them with STOKE and seeing if that was advantageous in terms of number of the CPU operations and achieved speed-up.

5.1 High Cyclomatic Complexity

In this subsection programs with high cyclomatic complexity will be tested. It is defined as a "metric measures the number of linearly independent paths through a piece of code"[1].

```
int fn(int x){
  if (x > 3) {
    if (x > 5) {
        return x - 1;
    }
    return x - 3;
  } else {
    return x + 3
  }
}
```

Figure 9: An example of a function with cyclomatic complexity of 3

Goal of these experiments is to check if STOKE is able to reduce number of conditional jumps and perform deletion of unreachable instructions.

5.1.1 Nested if statements

This program contains a function that takes an integer and consists of 18 if statements. The integer is changed inside the if statements and the different branches are chosen depending on the input value. The integer goes through this if statements one billion times.

`STOKE` managed to find a solution that has around half of the operations however it did not achieve a speed-up but the opposite. The time of program execution was longer from 8% to 13%.

5.1.2 Dead code

The function tested in this experiment looks very similar to the one tested previously. However, in this case, regardless of the input to the function it will always behave in the same way and return the same value. The aim of this experiment is to check if `STOKE` will manage to get rid of the unused conditional jumps and reduce the function to the most efficient form which is just returning zero.

`gcc -O3` managed to find such a solution, it computes XOR of the input register with itself which is equivalent to setting it to zero, and returns it. Unfortunately, the solution found by `STOKE` is not as straightforward. It also zeroes the return register with the first instruction. However, optimized code contains several unnecessary conditional jumps before finally returning the function. As could be expected those jumps make program execution slower than the `gcc -O3` version by at least 13%.

5.2 Random Memory Accesses

In this subsections programs that access certain parts of memory multiple times in a random order will be tested. Goal of these experiments is to check is how `STOKE` will handle frequent memory accesses and if it this will improve programs' performance over `gcc -O3`.

5.2.1 An array of an arbitrary size

This particular function takes a pointer to an array and three indices and then uses these indices to access an integer array pointed to by the first argument.

```
int f(int* arr, int i, int j, int k){
  int a = arr[i];
  a = a | arr[j];
  return a ^ arr[k];
}
```

Figure 10: Function accessing three different indices of an array with arbitrary size

Point of this experiment is to check if `STOKE`'s optimizations can make the program somehow access random memory addresses faster than program generated by `gcc -O3`.

```
# gcc -O3
pushq rbp
movq rsp rbp
movq rdi -0x8(rbp)
movl esi -0xc(rbp)
movl edx -0x10(rbp)
movl ecx -0x14(rbp)
movslq -0xc(rbp) rdi
movq -0x8(rbp) rax
movl (rax,rdi,4) ecx
movl ecx -0x18(rbp)
movl -0x18(rbp) ecx
movslq -0x10(rbp) rax
movq -0x8(rbp) rdi
orl (rdi,rax,4) ecx
movl ecx -0x18(rbp)
movl -0x18(rbp) ecx
movslq -0x14(rbp) rax
movq -0x8(rbp) rdi
xorl (rdi,rax,4) ecx
movl ecx eax
popq rbp
```

```
# STOKE
movl edx eax
vmovd ecx xmm6
vmovdqu ymm6 ymm15
movl (rdi,rax,4) ecx
xchgl eax esi
rorw $0xc0 cx
orl (rdi,rax,4) ecx
pextrq $0xfa xmm15 rax
xorl (rdi,rax,4) ecx
xorw $0x0 di
cmovpoq rcx rax
```

Figure 11: Function before and after `STOKE`'s optimization

STOKE managed to find a successful rewrite. Program with the optimized function achieved speed-up of at least 6%.

5.2.2 Small array

This function is quite similar to the one in the previous function. It also takes a pointer to an array and three indices and then uses these indices to access an integer array pointed to by the first argument. However in this case an array is of a fixed size of 10. The point of this experiment is to check if `STOKE` would behave differently with that small of a change and this particular class of a program.

```
int f(int arr[10], int i, int j, int k){
  int a = arr[i];
  a = a | arr[j];
  return a ^ arr[k];
}
```

Figure 12: Function accessing three different indices of an array with a size of 10

```
# gcc -O3
pushq rbp
movq rsp rbp
movq rdi -0x8(rbp)
movl esi -0xc(rbp)
movl edx -0x10(rbp)
movl ecx -0x14(rbp)
movslq -0xc(rbp) rdi
movq -0x8(rbp) rax
movl (rax,rdi,4) ecx
movl ecx -0x18(rbp)
movl -0x18(rbp) ecx
movslq -0x10(rbp) rax
movq -0x8(rbp) rdi
orl (rdi,rax,4) ecx
movl ecx -0x18(rbp)
movl -0x18(rbp) ecx
movslq -0x14(rbp) rax
movq -0x8(rbp) rdi
xorl (rdi,rax,4) ecx
movl ecx eax
popq rbp
```

```
# STOKE
movq rdi rax
xchgl edi edx
vpbroadcastw (rax,rdi,4) ymm0
bzhil edx ecx edx
cmovbl (rax,rdi,4) ecx
xchgq rax rdi
xchgl eax esi
orl (rdi,rax,4) ecx
xchgb al dl
xorl (rdi,rax,4) ecx
xchgl eax ecx
```

Figure 13: Function before and after `STOKE`'s optimization

`STOKE` managed to find a successful rewrite however the result of the run time measurements turned out quite differently. Both run times were around 15% faster than the programs from the previous experiments. However the program optimized by `STOKE` was around 4% slower than the program produced by `gcc -O3`. This shows that slight difference in the program may cause `STOKE` find a completely different rewrite for them.

5.3 High Computational Complexity

In this subsection programs with a significant computational complexity will be tested to check how `STOKE` handles functions containing multiple loops.

5.3.1 Travelling Salesman Problem

Traveling Salesman Problem is defined as follows "a permutation $P = (1i_2i_3...i_n)$ of the integers from 1 through n that minimizes the quantity a_{1i_2} where the $a_{\alpha\beta}$ are a given set of real numbers. More accurately, since there are only $(n-1)!$ possibilities to consider, the problem is to find an efficient method for choosing a minimizing permutation".[2]. This particular problem has various real-life applications and yet there is no efficient solution to it and the problem is considered NP-Hard. `STOKE` wasn't expected to find a faster algorithm but to optimize the existing naive one as any speedup of a program such inefficient would be significant.

```
int fxn(int s, int path[],
    int path_size, int graph[][V]) {
    int current_pathweight = 0;
    int k = s;

    for (int i = 0; i < path_size; i++) {
      current_pathweight += graph[k][path[i]];
      k = path[i];
    }
    current_pathweight += graph[k][s];
    return current_pathweight;
}
```

Figure 14: Function calculating distance of one of the possible paths

Program generated all possible permutations and then passes them to the function above to calculate its weight.

This experiment showed that `STOKE` is not production-ready yet. It managed to find potentially better solutions but some errors had to occur in the optimization process as the program produced a Segmentation Error in the fxn function after the original fxn assembly code was replaced by the one optimized by `STOKE`. Knowing `STOKE`'s limitations regarding functions containing loops, none or a small speed-up was expected but the fact that the program was not working at all was the only occurrence of that of all the experiments.

Different configurations were tried that produced different potential rewrites and the non-optimized version of the program was tested thoroughly and concludes that even though that these solutions passed all the test-cases generated by `STOKE`, they are not a guarantee of program safety.

5.3.2 Sum of three integers and matching brackets problem

These two experiments were described in a single subsection as they gave identical results.

The sum of three integers function takes a pointer to an array and a target value. Function consists of three for-loops which are trying to find three values with different indices that sum up to the target value. There exist solutions with complexity $O(n)$ but this particular algorithm is of complexity $O(n^3)$ The aim of this experiment was to check how `STOKE` would handle a solution to a problem which is not optimal.

Matching bracket problem is to given a string consisting of different types of brackets (in this case (,{ and [) find number of pairs brackets that are matched properly which is defined as that there is a opening and a closing bracket and between them there is either no other brackets or only other properly matched brackets. Solution used in the experiment used a stack data structure to efficiently calculate the solution as the algorithm has $O(n)$ complexity. The goal of this experiment was to check how `STOKE` would handle an efficient solution to a problem.

The results of `STOKE`'s optimizations to these problem were extremely disappointing. Despite trying multiple configurations of the search procedure it wasn't able to find a solution. Multiple cost functions, verification strategies, iteration timeouts, numbers of test cases were tried and all the search for better solution took combined around 24 hours after which it was decided that it would never be feasible to spend so much time on optimizing

such short functions. Hence, the experiment is considered as STOKE's failure and it was concluded that STOKE is not a always an appropriate Superoptimizer to deal with functions containing loops.

6 Responsible Research

6.1 Trustworthy results

Important part of this research is making sure that the provided results of the experiments are produced in a way that they will lead to correct conclusions. A lot of program time measurements were done to support certain statements. Performing them without taking measures against measurement bias could potentially make a lot of work worthless and conclusions wrong. To prevent that I consulted some of the best practices [4].

- Experiments were conducted in a minimal environment (freshly installed operating system).
- No 3rd-party applications were running in the background
- Stack starting address randomization was turned off on the machine.

6.2 Reproducibility

It was the goal from the beggining of this research for all the experiments could be reproduced on different machines because it might be the case that the results could vary e.g. on different processor micro architectures. To make that possible the code and STOKE's configuration files for all experiments written by me were put on the 4TU database[2]. All the experiments from the original paper were taken from the official STOKE repository[3] that was most recent on June 20, 2022.

7 Discussion

As it can be seen the results in different experiments vary. In the experiments taken from the original paper we can see that STOKE can satisfy the thesis that can produce programs at least as fast as the ones produced by `gcc -O3` (version 4.9) but the speed-up achieved on this particular computer was always less significant than those shown in the paper.

The results generated when identifying STOKE's ability to optimize different program classes do not seem as positive as the previous ones. A speed-up was achieved in only of these experiments, in the other ones slow-down was achieved ranging from 3% to 16%. These results are very disappointing, however it is important to keep in mind the limitations of this research. Even though quite a few configurations were tried for each experiment, a more experienced STOKE user (or a lucky one) might have written a configuration file that traversed the search space in a way that found a more efficient solution to the problem. It is important to also know STOKE's limitations in context of programs that contain loops, as mentioned in the original paper and confirmed here, that STOKE can't find the most optimal

[2]https://data.4tu.nl/articles/software/Programs_to_evaluate_superoptimizer_STOKE_/20099015/1
[3]https://github.com/StanfordPL/stoke

solution in these programs as it can only optimize loop-free programs so it tries to optimize the most inner part of the loops.

The most disappointing result was certainly one got when testing the problems with High Computational Complexity as Travelling Salesman Problem program in which after `STOKE`'s optimizations the program was caused a Segmentation Fault when executing the optimized function or two other problems for which it could not find a proper rewrite at all. There were also two experiments in high computational complexity for which `STOKE` did not find any suitable rewrite, which is very like to come from `STOKE`'s limitations regarding the code containing loops.

8 Conclusions and Future Work

The experiments confirmed that `STOKE` is able to produce programs at least as fast `gcc -O3` in some cases and the experiments from the original paper show that. However it is not true in all cases, `STOKE`'s limitations when it comes to programs that contain loops may cause a slow-down. `STOKE` is a tool that can generate programs which are significantly faster than the original solution, however it sometimes can do the opposite of that. It is inconvenient as creating a configuration file that makes `STOKE` generate efficient solutions is time consuming and takes a lot trials and errors and might be troublesome in big projects. It also can produce programs that crash as shown in the Travelling Salesman experiment.

Strong points	**Weak points**
It is able to propose novel solutions shortening the execution time and the binary size of a program	Struggles to find solutions or finds solutions slower than `gcc -O3` in case of programs with loops
It is able to reduce numbers of multiple logical and mathematical operations to a smaller amount decreasing number of processor cycles (e.g. p23 experiment)	Does not guarantee producing a correct solution
It is able to perform standard optimizations (e.g. removing dead code) with some limitations	Standard optimizations are not always performed in a perfect way (e.g. removing most of dead code but not all of it as described in section 5.1)
	Small changes to the configuration can cause `STOKE` to find a less optimal solution

Figure 15: Summary of `STOKE`'s strong and weak points

Although some of the experiments conducted in this paper have not reduced positive results the other ones show great potential that lies in Stochastic Optimization as some issues with `STOKE` are only characteristic for its implementation (e.g. accepting programs that SEGFAULT as a correct solution due to insufficient testing). `STOKE` is capable of finding really smart optimization as shown e.g. in p21 function experiment, however finding optimal

configurations of `STOKE` is difficult and takes a lot of work. Research on them and some general guidelines would be very useful for less experienced users. More investigation on `STOKE` itself also would be beneficial. It is important to investigate how `STOKE` could prevent producing programs that that crash and also in some cases produce programs slower than `gcc -O3`.

References

[1] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE Software*, 33(6):27–29, 2016.

[2] Merrill M. Flood. The traveling-salesman problem. *Operations Research*, 4(1):61–75, 1956.

[3] Henry Massalin. Superoptimizer: A look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5):122â126, oct 1987.

[4] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265â276, mar 2009.

[5] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *SIGPLAN Not.*, 48(4):305â316, mar 2013.