Small code with large binaries, final report

CHRISTOPHER TIBALDO (17-915-778)
MARCO HEINIGER (18-733-824)

When we generate assembly code, we can pass specific requirements to the compiler. These can range from wanting to generate fast, easy to debug, or even short assembly code (respectively the flags -O3, -Og, -Os). Our goal is to identify instances in which the goal of generating short assembly code, with the -Os flag, is not achieved. We thus look for C code segments in which the ratio between source and assembly code is excessive.

We propose a framework that through the use of automated code generation and reduction tools (respectively C-Smith and C-Reduce) is able to identify just such situations. Furthermore, we have iteratively created a set of test functions that are used as a target metric for the C-Reduce reduction process. These have been capable to identify multiple instances in which compilers, using the -Os flag, produce larger assemblies than what would be strictly necessary. These inefficiencies have been verified by comparing the outputs with different compiler flags, as well as that of other compilers.

1 INTRODUCTION

Compilers lie at the foundation of most modern software. Due to their prevalence, finding even small inefficiencies in their product can, cumulatively over all compiled programs, result in large time and energy savings. This project aims to automatically identify just such missed optimizations and generate code which illustrates these failures. Specifically we are looking for situations in which the the size of generated assembly code is unnecessarily large. Small binaries enable programs to be run with fewer cache accesses as well as being able to better run on computationally constrained embedded devices, whose prevalence is ever-increasing in our highly interconnected world.

We have created a framework which uses automatically generated C code to perform a reduction process. Code which doesn't contribute to excessive code bloat gets gradually removed until a minimized example, which still displays this undesired behaviour, can be provided. We have been able to identify and reduce to a "smallest working example" multiple instances for which the GCC compiler created unnecessary assembly instructions. The results have been verified by inspecting the output by hand and comparing it with the output of the CLANG compiler. On top of finding missed optimizations in GCC we believe to have created an easily extendable framework to identify failed optimizations within compilers. This might further be used to compare multiple compilers, flags and versions among each other.

2 APPROACH

Finding instances where large binaries are generated by small sections of C code, when the -Os flag is in use is an exploratory task. This can be tackled by building an automated pipeline based on multiple existing libraries. The general idea for this process can be seen in the image below.

The first step in the pipeline is to generate code which we use as our basis for the following steps. Starting from this initial code we calculate a binary-to-code-length ratio. Afterwards, we reduce this code, while trying to keep the output characteristics unchanged. As reduction of code can insert undefined behaviour as well as compiler issues, we perform a sanitization check of the newly reduced code. If the sanitization failed we revert to the previous result and retry the reduction from there. If on the other hand the code is valid we compute the new ratio of the reduced program as well

Authors' addresses: Christopher Tibaldo (17-915-778), tibaldoc@student.ethz.ch; Marco Heiniger (18-733-824), mheinig@student.ethz.ch.

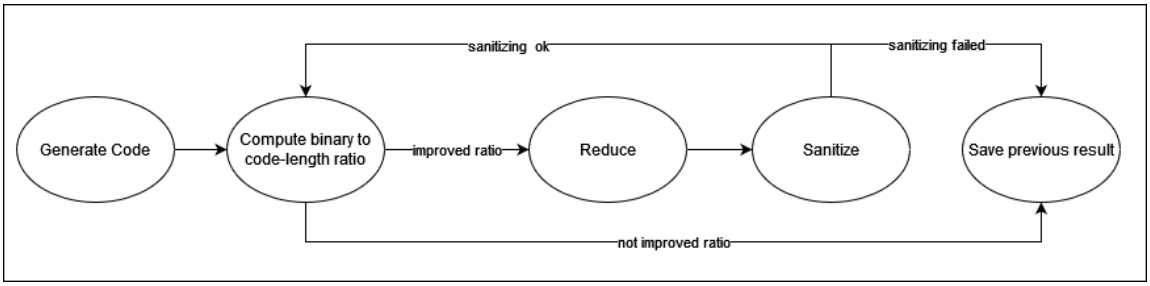

Fig. 1. Outline of the reduction process

as check that any additional constraints are still respected. If the new ratio has increased we keep the new code. The reduction step is applied again until no further improvement can be made.

3 IMPLEMENTATION & RESULTS

The concrete implementation of the project was performed using the python framework Diopter[1]. This framework provides easy access to the following components mentioned in the previous chapter 2:

- **Code generation:** For the code generation we used Csmith which Diopter provides an interface to. We used this to generate the initial C code on which we afterwards performed the reduction operation. Since Csmith offers a wide variety of flags to customize the code it generates, we decided to enable or disable these with a 50% likelihood. To provide an indication of the flags used by Csmith for further analysis, we have decided to create a *CSMITH_parameters.txt* file for each run. Moreover, the list of all available flags can be found in appendix A.
- **Code reduction:** Reducing the code was performed by C-Reduce, a powerful code-reduction tool, which attempts to remove sections of code which allows it to optimize the result of an "interestingness function". The interface, provided to us by Diopter, allowed us to pass to C-Reduce a "ReduceObjectSize" class. This class contains a self-written test function which gives a binary answer to the interestingness of the code generated by C-Reduce. More information about the test function is given further below.
- **Code sanitization:** A sanitization checker which ensures that generated code compiles correctly. This check is run, whenever new code is generated as well as after each reduction step. Using GCC, the sanitizer makes sure that no errors, such as "division by zero or "invalid in C99" occur in the output.

The usual application of C-Reduce is to identify the specific segment of code which causes a bug. In this case, the test function would be the existence or absence of the bug in the program's output. In our case, the interestingness function primarily checks if the ratio (binary size / code length), when compiled with the -Os flag, has increased compared to previous steps in the reduction process. To be more precise when computing the ratio, we used the following formula $\frac{binaryLength - emptyAssemblySize}{codeLength}$, which ensures that we don't consider lines of assembly which are trivially necessary for any C program, where $emptyAssemblySize$ = int main(){}.

We have noticed that with the trivial code ratio computation approach the generated code is not very interesting, since it mostly consists of very few lines. We have thus proceeded with designing further tests, in which we added different kinds of constraints to C-Reduce, such that the output is of more interest. Designing these tests was performed in an incremental/experimental way, by observing generated code from previous tests.

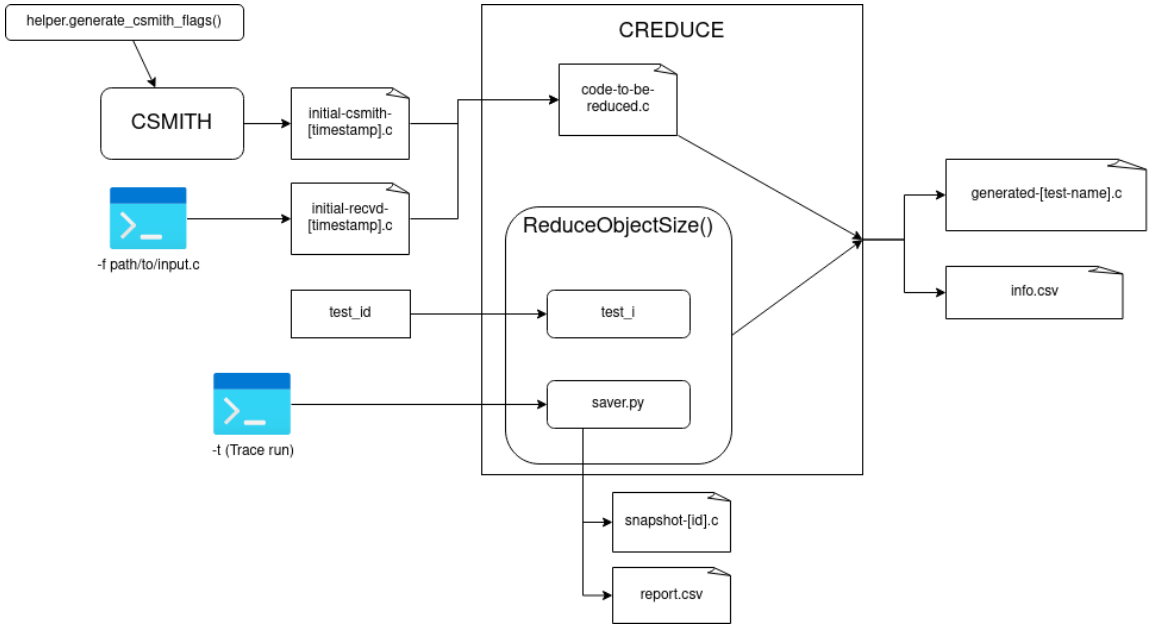

Fig. 2. Schematic representation of the reduction process in initial-reduction.py

In the following sections, we want to illustrate what the ideas behind these tests are and what results we obtained. Their names as used in code have been added in parenthesis.

3.1 No restrictions (test 0)

This was the starting point in which we only observed the ratio of the generated code. We noticed that the generated C code tends to be comprised of very few lines, as already mentioned. We were able to find a few bugs using this method, which are further described in section 4.

3.2 Minimum number of lines (test 1)

For this test, we added the constraint of a minimum amount of lines required in the generated output. Our reasoning for this was that with a lower bound on the amount of lines we could generate more complex outputs and while the ratios obtained might not be as extreme as those achieved in test 0, we might be able to identify more subtle failed optimizations. The problem with this approach was that the code returned by C-Reduce simply contained a lot of either empty lines or lines only containing ";" and "{" symbols. As such results are not very interesting, we thus decided to further elaborate our test function.

3.3 Using AST to set line limit (test 3)

In the previous test, we utilized lines of code as a proxy metric to approximate the "amount of code" we wanted to keep. We realized what we are actually interested in were the "Logical lines of code". The most convenient method we identified to approximate this is by counting the number of abstract syntax tree (AST) nodes contained in the generated code. To obtain a text representation of the AST, we have used the "clang -ast-dump" command. Based on this command,

we have built the "ast-parser.py" file, which parses the output of the clang command and returns a python tree structure. Having this tree structure allowed us to perform a series of interesting operations, like weighting specific nodes more, or removing others entirely. In this test, we decided to remove "NullStmt" nodes (which represent lines of code only containing ";"), as well as set a minimum number of AST nodes to 30. With these requirements, we got most of the time code with many variable definitions/declarations and unused functions.

3.4 Using AST to count lines of code (test 4)

In parallel to using the AST to set the line limit, we decided to compute the ratio of the program not by using the length of the generated C code but by the number of AST nodes and also by setting the minimum number of AST nodes to 30. The reasoning was that for two identical programs, if one were to use variables with more extended names, it would give a better ratio than by using more abbreviated names. We decided such a difference does not make the code interesting, since a compiler would get rid of user-given names. But as with the previous test, we got unused functions and variable definitions/declarations. This leads us to the fact that we further had to elaborate on the unused functions/variables.

3.5 Removing unused functions (test 5)

As already mentioned in subchapter 3.4 we noticed numerous unused functions and variables in the results of our tests. Since these tend to be automatically removed by the compiler, we have chosen to eliminate them. This was performed by parsing the warnings outputted from the "clang -Wunused" command and successively removing these functions from the Abstract Syntax Tree of the program. In addition, to this test, we set a minimum number of AST nodes to 30. The problem we confronted was that this modification had no impact on our results (further explanation in subchapter 3.9).

3.6 Annotating code as static (test 6)

Another way to avoid these unused variables and functions was by using a static annotator [2] to mark all global variables as "static". This enables the compiler to perform more optimizations since it does not have to take into account other processes which might interact in an unexpected way with these variables. Moreover, when we compute the ratio we use the same function as described in subchapter 3.4. The results were by far the most promising, and we also discovered some failed optimizations with it (4).

3.7 Increasing weight of undesired statements (test 7)

This test illustrates a method that might be used in a more interactive version of this program. It disincentivizes nodes of a specific kind within the AST. In our example, the disincentivized nodes are: for loops, printf statements, and multiple kinds of declarations. By assigning an excessive weight to these nodes (we add multiple dummy nodes for each of the undesired ones) we can ensure the reduction process gets steered away from generating code containing these features. They might still occur if their presence enables a significant increase in the obtained ratio.

3.8 Possible future work (test 8, test 9)

The primary goal of these tests is to identify failed optimizations of -Os compared to -O3, as well as finding instances in which the CLANG compiler performs worse than GCC. We have been able to obtain interesting results from these tests

19, but due to time constraints as well as their inconsistent running behaviour we have decided to mark them as "future work".

3.9 Comparison of various approaches

After representing all various approaches, an adequate comparison is needed regarding the ratio and how interesting code the different tests created. As a first comparison, figure 3 shows how the ratio changed during the reduction process. We note that among all our runs the reduction profile stays remarkably consistent.

The fastest and largest reduction is archived by the first test, which only checks if there has been an increase in ratio. Most probably, C-Reduce can radically apply its reduction techniques without getting penalized, as removing any code increases the ratio. The next best test is where we included that a minimum number of lines have to be obtained. It is clearly visible C-Reduce takes a longer time and reaches a worse ratio than with no restrictions. But in contrast to the other tests, the amount of time used to generate the result was quite moderate. For both tests on a normal laptop, a result can be expected at a maximum of 30 minutes each. We note that even though a really high ratio is achieved in these tests, the results are often uninteresting. They mostly consist of only an empty main function.

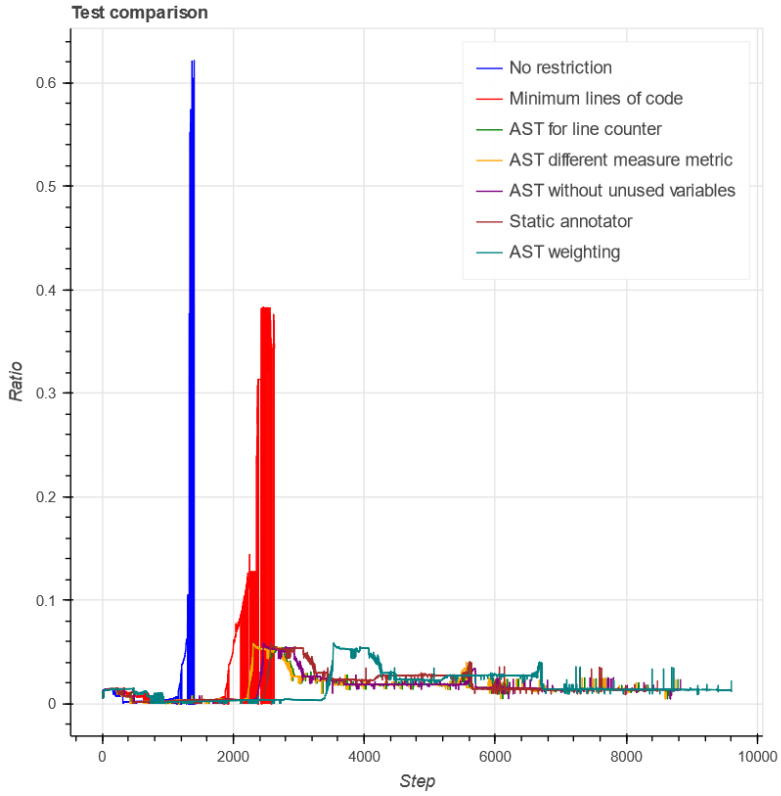

Fig. 3. Calculated ratios over timesteps of all tests

For the other tests where we use the AST in some form, we have a massive increase in the reduction process runtime, number of steps as well as a significantly smaller reached ratio. The reduction process with one of those tests required

a minimum of one hour on a normal laptop, depending on how fast C-Reduce finds a suitable solution. A more detailed comparison is visible in figure 4. Out of almost all runs the test where we use the AST as a line counter, where we use it as metric, and where we use it to remove unused variables/functions, obtained the same result and thus also the same ratio. The difference between them was how fast they reached this result. For e.g. the test which removes unused variables/functions needs the most amount of time out of these 3 mentioned. This becomes apparent when inspecting our code, as this test requires several steps for the evaluation. The question that now arises is why all 3 tests almost consistently generate the same result. We believe our designed test functions are not enough distinguishable for C-Reduce and that it uses the same passes for all test cases. To a great extent, this also provides a firm indication that C-Reduce is deterministic in the way it operates.

Similarly, also with the static annotator test and with the test where specific nodes are disincentivized, identical results were generated during our runs. Compared to the other AST tests except in a few instances the ratios were worse and by figure 4 it is evident that runtimes were longer. The increased required time for the reduction process represents a signal that C-Reduce used other passes and that C-Reduce had more difficulties finding a better solution. One reason is that with the static annotator, the resulting Assembly is only changed when functions or variables are effectively used in the main function. Hence, C-Reduce has to perform numerous specific alterations to increase the obtained ratio.

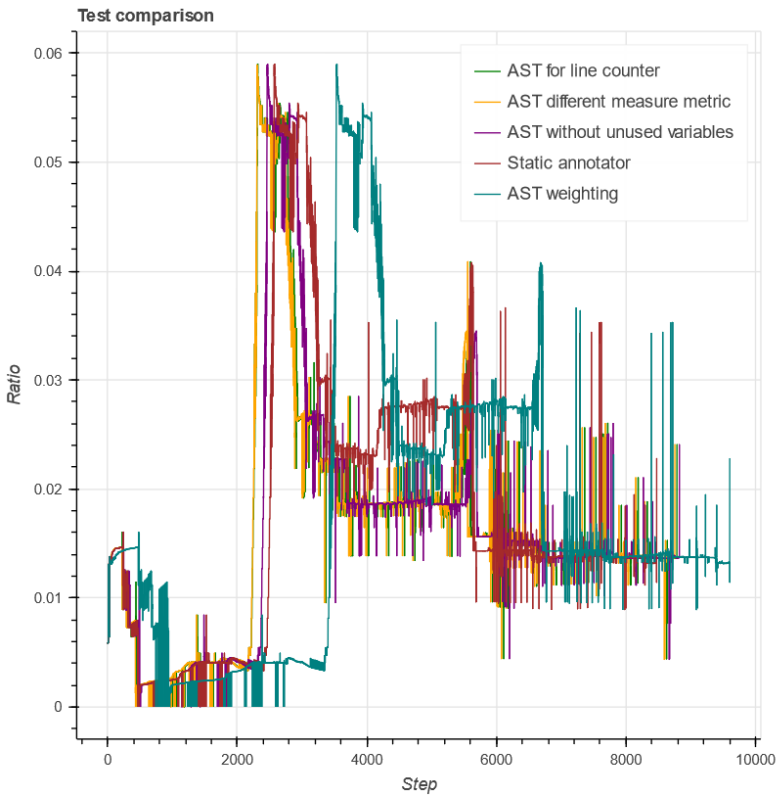

Fig. 4. Calculated ratios over timesteps of selected tests

3.10 Saving intermediate results

Besides having implemented various test functions, we have included an optional saving feature in our test runs. When given the flag -t (as in trace run) to initial-reduction.py, we save the size of the attempted reduction at each step as well as a snapshot of the code every 100 reduction steps. This feature allows us to see in real-time how exactly the reduction is proceeding. On top of this, the generated snapshots might be useful if in future work the reduction process is made more interactive. We can already take these generated snapshots and pass them as inputs to initial-reducer.py via the -f flag, instead of running the reduction process on the generated Csmith code. We can also manually enable/disable the selected tests but believe that with a more streamlined process, it could be possible to adapt the test on the fly to dynamically steer the reduction process in a desired direction.

3.11 Executing the project on your machine

All the code presented in this project can be found in the following git repository:
https://gitlab.ethz.ch/tibaldoc/ast_largebinaries_smallcode
We have also created a custom docker image combined with a docker-compose file. This allows anyone to easily and consistently run our program by following the instructions outlined in the README.md provided in the repository. We have used this configuration to obtain outputs for multiple runs on a remote server, using the "utilities/multi_run_manager.py" script. The script "utilities/identify_interesting.py" was then used to identify interesting generated code. The results of this process can be found in the "artifacts" branch on GitLab.

4 FAILED OPTIMIZATIONS IN GCC

Following are some of the failed optimizations we were able to find with the generated results.

```c
#include <stdio.h>
int a;
int main() { printf("", a); }
```

Fig. 5. C code of first failed optimization

```
.LC0:
        .string ""
main:
        push    rax
        mov     esi, DWORD PTR a[rip]
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        call    printf
        xor     eax, eax
        pop     rdx
        ret
a:
        .zero   4
```

Fig. 6. Assembly code with GCC 13.1 -Os

```
main:
    xor     eax, eax
    ret
a:
    .long   0
```

Fig. 7. Assembly code with Clang 16.0.0 -Os

When looking at the code in figure 5, the expected output should be an empty function call, as in the printf statement nothing is written and the variable *a* is unused. GCC is unable to optimize this statement, as visible in figure 6. Clang, on the other hand, recognizes this thanks to the LLVM Opt Pipeline step *InstCombinePass*. By checking older versions of GCC, as well as using different flags (-O1, -O2, and -O3) this optimization failure is consistent. Clang is also consistent with its optimization. The failed optimization also occurs when the variable *a* set to static.

The second failed optimization we found was with test 6 which makes everything static. Now when deleting all unnecessary variables and functions outside the main function (which are optimized anyway by the compiler), we get the code visible in figure 8.

```c
static int i;
int main(void) {
    for (;i < 7;i++) {}
}
```

Fig. 8. C code of second failed optimization

```
main:
        mov     edx, DWORD PTR i[rip]
        xor     ecx, ecx
        mov     eax, edx
.L2:
        cmp     eax, 6
        jg      .L9
        inc     eax
        mov     cl, 1
        jmp     .L2
.L9:
        test    cl, cl
        je      .L4
        mov     eax, 7
        xor     ecx, ecx
        sub     eax, edx
        cmp     edx, 7
        cmovg   eax, ecx
        add     eax, edx
        mov     DWORD PTR i[rip], eax
.L4:
        xor     eax, eax
        ret
```

Fig. 9. Assembly code with GCC 13.1 -Os

```
#-------- GCC O3 --------------------------
main:
        cmp     DWORD PTR i[rip], 6
        jg      .L2
        mov     DWORD PTR i[rip], 7
.L2:
        xor     eax, eax
        ret

#-------- CLANG Os -------------------------

main:
        cmp     byte ptr [rip + i], 0
        jne     .LBB0_2
        mov     byte ptr [rip + i], 1
.LBB0_2:
        xor     eax, eax
        ret
```

Fig. 10. Assembly code with GCC 13.1 -O3 and Clang 16.0.0 -Os

Here the expectation would be that the generated Assembly code is going to be extraordinarily small as the body of the loop is empty. Now by comparing the produced Assembly of GCC (9) and Clang (10) a significant difference is notable. Once more, by inspecting the LLVM Opt Pipeline of the Clang compiler, the most important steps are the

LoopSimplifyPass on main and the *LoopDeletionPass on for.inc* which reduce the Assembly to a minimum. GCC is unable to apply such optimization steps with the -Os flag as Clang does. As with the first failed optimization, we investigated further and checked if other flags show a similar failure. Surprisingly, with -O2 and -O3 (10) the generated Assembly code is practically identical as with Clang (10). This is interesting, as the purpose of the -Os flag is to reduce the resulting binary size by applying various optimizations, whereas with the -O3 flag also an enlargement of the binary could be expected.

A further failed optimization with test 1, which ensures that the ratio improved and a minimum number of lines of code are obtained. The code 11, after removing automatically optimized parts, is comparatively simple as it sets the value of the variable to zero. But when analyzing and comparing figure 12 and 13, with GCC 13.1 -Os an unnecessary step in the beginning is visible. It initially makes an *xor* which sets the value to 0, and then it makes the *mov* instruction. Whereas in figure 13 with Clang -Os and GCC -O3 directly, the *mov* instruction is applied. An interesting observation is that this failure only occurs when setting the value to zero. Further, this failure seems to be introduced recently, as it solely occurs from version 12.1 upwards. All previous versions of GCC with the -Os flag are performing the same optimization as in figure 13.

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

Fig. 11. C code of third failed optimization

```
main:
        xor     eax, eax
        mov     DWORD PTR a[rip], eax
        xor     eax, eax
        ret
a:
        .zero   4
```

Fig. 12. Assembly code with GCC 13.1 -Os

```
#-------- GCC O3 -------------------------
main:
        mov     DWORD PTR a[rip], 0
        xor     eax, eax
        ret
a:
        .zero   4
#-------- CLANG Os -----------------------
main:
        mov     dword ptr [rip + a], 0
        xor     eax, eax
        ret
a:
        .long   0
```

Fig. 13. Assembly code with GCC 13.1 -O3 and Clang 16.0.0 -Os

The last failed optimization for GCC we detected was with the test 6, which set everything as static. Figure 14 represents the code that causes the failure. It is a relatively complex usage of pointers and value assignments, but the crucial part here is that no variable is reused in this code. Following, the expected Assembly should look like figure 16 as every assigned variable/pointer is static. GCC (15) on the other hand generates the exact Instructions to set the value of the memory location of *g_390_0_0* to zero. A way to explain the difference between GCC and Clang is to examine the LLVM Opt Pipeline. LLVM executes a live variable analysis (LVA) of the code, which exploits the dead code of the code in figure 14 [4]. Presumably GCC, in this case, has a problem with its LVA as they are also using live variables [5].

Previous versions of GCC and different flags result in identical failures, which could indicate that this problem has not been discovered yet.

```c
static int g_8 = 0x1EB2391L;
static int *g_390_0_0 = &g_8;
static int **g_389 = &g_390_0_0;
int main() { *g_389 = 0;}
```

Fig. 14. C code of fourth failed optimization

```
main:
        xor     eax, eax
        mov     QWORD PTR g_390_0_0[rip], rax
        xor     eax, eax
        ret
g_390_0_0:
        .quad   g_8
g_8:
        .long   32187281
```

Fig. 15. Assembly code with GCC 13.1 -Os

```
main:
        xor     eax, eax
        ret
```

Fig. 16. Assembly code with Clang 16.0.0 -Os

4.1 Failed optimizations with Clang

Although our main focus was not to find failed optimizations with Clang, we discovered in a successful run of test 9 in which we check if the ratio with the GCC compiler is better than with the Clang compiler, one failed optimization. The code in figure 17 performs a bitwise and of $4095_{10} = 111111111111_2$ and $255_{10} = 11111111_2$. As compilers can do precomputations the expected c code, should be: static int a = 255; int main (). The subsequent step of the compiler should be to recognize that variable *a* is unused and thus can be optimized away. Unluckily, both compilers (figure 18 and 19) are incapable to perform these steps, and Clang is doing even worse than GCC in this case.

```c
static int a = 4095;
int main() { a = a & 255;}
```

Fig. 17. C code of fifth failed optimization

```
main:
        and     DWORD PTR a[rip], 255
        xor     eax, eax
        ret
a:
        .long   4095
```

Fig. 18. Assembly code with GCC 13.1 -Os

```
main:
        movzx   eax, byte ptr [rip + a]
        mov     dword ptr [rip + a], eax
        xor     eax, eax
        ret
a:
        .long   4095
```

Fig. 19. Assembly code with Clang 16.0.0 -Os

5 RELATED WORK

The fundamental process underlining this project is that of delta debugging. Formally introduced in 1999 by Andreas Zeller in his seminar paper "Yesterday, My Program Worked. Today, It Does Not. Why?" [9], lather further expanded in "Isolating Failure-Inducing Input"[10]. The goal of this process is to isolate the source of errors among large inputs. An essential operation when dealing with code as complex as that processed by compilers. On these principles the C-Reduce tool has been built, specifically to perform this reduction task on C and C++ code [7]. The advantage of such a tool are that it can operate in a context aware fashion, which allows it to generate extremely concise and syntactically correct pieces of code, which still trigger these errors.

A challenging task for compiler designers is to provide generic tools, which satisfy the broad requirements of their users. Compiler flags allow programmers to specify what requirements the compiled code should satisfy. There have been multiple analyses of the effects these flags have on the produced assembly size[3, 8], as well as how automatically constructing them might help in getting better suited assembly code[6]. Our project helps in highlighting instances in which optimizations promised by specific compiler flags are not performed.

6 CONCLUSION

We have been able to achieve our initial goal of creating a tool to automatically identify binaries for which compilers miss possible optimizations for binary size. On top of this we have created a framework on which future research on this topic might be conducted. We have shown how different testing functions affect what kind of code is found and we have provided concrete examples for which the latest version of GCC fails to perform proper binary size optimization. Following are some of the limitations in our current work, which might be addressed in future projects: The current project focuses mostly on analyzing the binary size of GCC -Os. We believe that the approach taken can be extrapolated to analyze the results of different compilers, as well as investigating the effect of more specific flags used by them (such as -finline-small-functions, -fmerge-all-constants, ...). Using custom input code, taken for instance from online GitHub repositories, instead of using tools to randomly generate it might help in highlight inefficiencies which are widespread in real world code. There would thus be a greater impact when fixing these.

7 FUTURE WORK

We believe to have created a quite robust framework which enables the investigation into inefficiencies in the compiler's binary size optimization process. These are possible directions for future projects along the same lines:

(1) Interactive reduction, using the snapshots we save as "stopping points" and letting the user adapt the test function based on which features they want to keep/reject (test 7 already penalizes specific functions, but it currently does this for an entire run, instead of dynamically for a subsection)
(2) Adding tests for differences in compiler output for different flags (Os/O3). Find instances where O.. creates less assembly than Os. (A trial implementation can be found in reduction_functions.py test_8)
(3) Add tests to find different optimizations by different compilers (A trial implementation can be found in reduction_functions.py test_9)

A CSMITH FLAGS

The list of flags randomly enabled/disabled when generating code with Csmith:
default_options_pool = ["arrays", "bitfields", "checksum", "comma-operators", "compound-assignment", "consts", "divs", "embedded-assigns", "jumps", "longlong", "force-non-uniform-arrays", "math64", "muls", "packed-struct", "paranoid", "pointers", "structs", "inline-function", "return-structs", "arg-structs", "dangling-global-pointers",]

REFERENCES

[1] 2023. DeadCodeProductions/Diopter. https://github.com/DeadCodeProductions/diopter. [Online; accessed 23-May-2023].

[2] 2023. Static C code annotator. https://github.com/DeadCodeProductions/static-globals. [Online; accessed 23-May-2023].

[3] Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*. 1–9.

[4] Jean-Claude Fernandez, Marius Bozga, and Lucian Ghirvu. 2003. State space reduction based on live variables analysis. *Science of Computer Programming* 47, 2 (2003), 203–220. https://doi.org/10.1016/S0167-6423(02)00133-8 Special Issue on Static Analysis (SAS'99).

[5] GCC. 2023. Liveness information. https://gcc.gnu.org/onlinedocs/gccint/Liveness-information.html. [Online; accessed 30-May-2023].

[6] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 165–174.

[7] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[8] Anderson Faustino da Silva, Bernardo NB de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the space of optimization sequences for code-size reduction: Insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 47–58.

[9] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[10] Andreas Zeller. 2001. Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering. Citeseer* (2001).