Dead Code Insertion with Variable Conditions

COLIN ARNET & JULIAN NEFF

Missed compiler optimizations reveal performance bugs in compilers. Using dead code elimination (DCE) Theodoridis et al. [8] were able to detect performance bugs by inserting "optimization markers" into randomly generated C programs and compiling them with different versions and options. In this paper we extend this approach by inserting dead code into the programs by adding unsatisfiable if-statements. We performed different experiments with variations of our method. Each experiment either adds additional markers to each basic block and/or keeps track of local and global or only global variables only. With each experiment we were able to detect interesting cases which dead was not able to. In one particular experiment our approach was able to detect 33% more cases than dead. We hope our extension leads to a better understanding in the analysis of DCE.

1 INTRODUCTION

Compiler optimizations are fundamental in improving program performance. They also introduce great potential for hard to find bugs. For example, the correctness of a program can be disturbed by incorrect optimizations. In addition to correctness bugs, optimizations can also be missed, resulting in performance bugs. In this paper, we will focus on generating cases that reveal performance bugs.

Dead Code Elimination (DCE) is an important optimization technique that removes unnecessary statements that are never used during program execution. This makes the final binary smaller and faster to execute. For this, many different methods such as constant propagation and reaching definitions are used to detect dead code. Because DCE is closely connected to other optimization methods it is a good indicator of the optimization quality. Therefore, it is beneficial to inspect DCE optimizations to get insight into missed optimization opportunities. For this, Theordoridis et al. [8] introduce `dead`, a tool that detects missed optimizations in DCE.

`dead` inserts Markers into every basic block and then compiles a randomly generated program with different compiler versions and optimization levels. Then it checks which markers are still alive in the binary and then asserts if a case is interesting. The interestingness is determined by the difference in alive markers. For example, if a program compiled with a lower optimization level has less alive markers as the higher optimization level, there is a performance bug in the higher optimization level. The same goes for older versions or different compilers (`gcc` and `clang`).

`dead` uses `CSmith` [11] to generate random C programs. These programs are not built with specific dead code situations in mind. For this, we extend `dead` to insert dead code into the `CSmith` programs to get more interesting cases with the goal to reveal more performance bugs. There are different variants of dead code that can be inserted. Unused expressions and variables are examples of dead code. Unreachable basic blocks are another category of dead code. They can be defined by execution paths that have unsatisfiable path conditions. In this paper we insert unreachable blocks into the programs. To achieve this we insert if-statements that contain a unsatisfiable conditions that are based on the variable values that are reached during the program execution.

Author's address: Colin Arnet & Julian Neff, carnet@student.ethz.ch,neffj@student.ethz.ch.

With our approach we modify the programs generated by `CSmith` to get programs that are more interesting for the `dead` tool. This way `dead` can detect more interesting cases and therefore catch more performance bugs.

In Section 2 we describe our approach in more detail and explain the implementation in Section 3. The results are discussed in section 4. Related Work is presented in Section 5 and we conclude this paper in Section 6.

2 APPROACH

In this section we will describe our approach in more detail. The goal of this is to insert dead code into randomly generated C programs. We chose to insert dead code by creating basic blocks that are unreachable.

2.1 dead

`dead` is the tool created by Theodoridis et al. in their paper [8]. It fuzzes different compiler versions and options to catch performance bugs. For this the tool first lets `CSmith` generate random C programs. The programs are generated in a way that they don't introduce undefined behavior and are guaranteed to terminate. These `CSmith` programs were used extensively to test compilers [11].

After the generation of the random C program, `dead` takes the program and instruments it for its use. To prepare the program for use, `dead` inserts empty functions into every basic block of the program. These functions are called `DCEMarker()`. The code examples 1 and 2 show how the markers are set. After instrumenting the program, it is checked if it is interesting. A program is interesting when it shows a possible performance bug. For this it is compiled with different compiler versions and optimization levels that are provided as arguments. Then the presence of the markers is then analyzed. If a marker is present, it is called "alive". A performance bug is detected when a newer version or higher optimization level has "alive" markers that aren't "alive" in older respectively lower optimization levels. In the case of a difference in the compiler version, this would mean that the newer version did not catch an already known optimization opportunity. The same goes for the optimization levels, where a higher level does not catch an optimization done in lower levels. If such a case occurs the initial program is considered interesting. In a further step this program can be used to report a performance bug in the compiler. With `CReduce` [6], the program is reduced to the essential parts that trigger the bug. This helps to make the bug report easier to understand.

Out of 10,000 `CSmith` programs `dead` was able to catch 2,637 missed optimization opportunities in `gcc` and 318 in `llvm` [8].

Listing 1. Original Code; C Program before instrumentation

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    // true if num is divisible by 2
    if(num % 2 == 0){
        printf("%d is even.", num);
    }
    else {
        printf("%d is odd.", num);
    }
    return 0;
}
```

Listing 2. Instrumented Code; C program after inserting DCEMarkers

```
#include <stdio.h>
void DCEMarker1_(){};
void DCEMarker2_(){};
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    // true if num is divisible by 2
    if(num % 2 == 0) {
        DCEMarker1_();
        printf("%d is even.", num);
    }
    else {
        DCEMarker2_();
        printf("%d is odd.", num);
    }
    return 0;
}
```

2.2 Extension

Except from inserting markers dead does not modify the randomly generated CSmith programs. Originally, the CSmith programs are generated to test compilers without focusing on optimizations. This means that CSmith does not generate programs with tricky DCE situations in mind. To cover this, we extend dead in such a way that it inserts dead code into the programs. We generate dead code by inserting unreachable basic blocks that are guarded by if-statements with unsatisfiable conditions.

To generate unsatisfiable conditions we run the program and keep track of the variable values. Each variable var has a set of values $vals = \{val_1, val_2, ..., val_n\}$ that the variable assumes at some point in the runtime. For each variable there is also a val_x that is never reached during runtime. This means that $val_x \notin vals$. So in any point during the execution $var == val_x$ will never evaluate to TRUE. $var == val_x$ can be taken as part of the unsatisfiable condition. In our extension, we generate the unsatisfiable condition using this approach.

Our extension inserts the conditions around the markers. At each marker there is a set $vars = \{var_1, var_2, ..., var_m\}$ of live variables. This includes the local variables and the global variables. For each var_i we have a set $vals_i$ that contains all values reached by var_i during execution. As we have seen before, for each variable var_i, we can make an unsatisfiable condition $var_i == val_{i\_x}$. All conditions together give the following condition: $var_1 == val_{1\_x} \vee var_2 == val_{2\_x} \vee ... \vee var_m == val_{m\_x}$. Since every part of the formula is unsatisfiable, the entire formula is unsatisfiable. Note that we use the OR operator. This is because for the AND operator only one part of the formula has to be unsatisfiable for the whole formula to be unsatisfiable. With OR every single part has to be unsatisfiable for the entire formula to be unsatisfiable. This makes it harder for the compiler since it will have to check each part for its unsatisfiability to make the optimization. In the end the code would be transformed into something like this 3:

Listing 3. Instrumented Code with our extension

```
// from:
DCEMarker0_();
// to:
if(var1 == x1 || var2 == x2 || varm == xm) {
    DCEMarker0_();
}
```

3 IMPLEMENTATION

In this section we will describe how the approach was implemented.

Our implementation intercepts the instrumented programs from dead. They have the same form as shown in the code example 2. The programs are then modified by our dead code inserter and then given back to dead to do the interestingness analysis. The main idea for our dead code insertion is to use gdb to run the program step by step and obtain the variable values throughout the run-time. To achieve this, the input programs have to go through different steps.

First, we have to generate a command file that contains all the commands that are run directly after gdb is launched. We search for all markers in the program and set breakpoints for each marker. Then we get all global variable names and set the print commands for each global variable. To get the values of the local variables that are alive at this point, we use the `info locals` command which prints all variable names and values that are currently in the stack. With a loop, the print commands are repeated for each breakpoint until the program terminates. The output is stored in a log file that is processed later. The program then has to be compiled without optimizations to then run gdb on it with the generated command file.

In the next step the log file is analyzed. For each breakpoint in the execution, it contains the variable names and values. The data is parsed and stored. With the gathered data, the unsatisfiable condition is generated. From the log file analysis, we know the variables and values that are reached during execution at each marker. For each variable we generate a random value and check if the value is reached during runtime. If this is true, a new random value is generated and tested. If the random value is not reached by the variable, we can generate a part of the condition by checking the

equality *var_name == random_value*. We do that for each variable and chain them together with the OR operator.

In the final step we insert the generated condition into the code with a if-statement and put the marker into the newly generated code block. After this we return the new code to dead which handles the interestingness analysis.

It is possible to add more modifications to the code with different approaches.

4 RESULTS

4.1 Experiment Setup

To evaluate our extension, we analyzed the number of interesting cases generated. For this we let CSmith generate programs and dead insert the DCEMarkers. Then we check the interestingness with and without our extension and store the gathered data for statistical analysis. We performed multiple experiments with different variations of our method. For each variation, we generated 5,000 cases. For the interestingness analysis we used 5 versions of gcc. Detailed version numbers are listed in section 7.3.

4.2 Variations

For our extension, we created four variations. They can be divided into two categories. The first category uses **no additional markers** and works as described in section 2. The second category uses **additional markers**. It inserts a new marker after the basic block guarded by the unsatisfiable if-statement as can be seen in the code example 4. This way we try to keep the properties of the original marker positions.

Listing 4. Instrumented Code with our extension with additional marker

```
// from:
DCEMarker0_();
// to:
if(var1 == x1 || var2 == x2 || varm == xm) {
    DCEMarker0_();
}
DCEMarker1_();
```

The two categories were tested by keeping track of the values of the local and global variables and also by keeping track of only the global variables. The results are presented in the next subsections.

4.3 No additional marker; Only Global variables

In this setting, we only kept track of global values without additional markers. Of the 5000 test cases, we found 134 interesting cases. Out of these 56 (41.8%[1]) were uniquely found by our approach while dead was "only" able to find 29 (21.6%) cases which our approach was not able to detect. There were also 49 (36.6%) cases which were found by both. This is the only setting in which we found significantly more cases than dead. The detailed results of this setting can be seen in the first column of Figure 1.

[1]The percentages are in relation to the number of interesting cases as can be seen in figure 2

4.4 Additional marker; Only Global variables

Now adding additional markers, as previously described, reduces the number of interesting cases to 103. Of these, our approach was only able to find 30 (29.1%), which is about halve as much as before. Interestingly, the number of cases found by both approaches remained proportional to the previous one with 38 (36.9%). Looking at dead we see that under this setting it achieves slightly better results as it is now able to find 35 (34%) unique cases. The detailed results of this setting can be seen in the second column of Figure 1.

4.5 No additional marker; Local and Global variables

Removing the markers again but keeping track of local and global variables reduces the number of interesting cases to 77. Our approach struggles under these circumstances as it found only 4 (5.2%) unique cases while dead still again improved its performance by finding 43 (55.8%) unique cases. In addition, there were 30 (39%) interesting cases that both approaches found. The detailed results of this setting can be seen in the third column of Figure 1.

4.6 Additional marker; Local and Global variables

At last we look at the setting where we apply additional markers and keep track of local and global variables. In total there were 82 interesting cases found. Out of these 13 (15.9%) were uniquely found by our approach while dead found 45 (54.9%) unique cases. The remaining 24 (29.3%) cases were detected by both approaches. The detailed results of this setting can be seen in the fourth column of Figure 1.

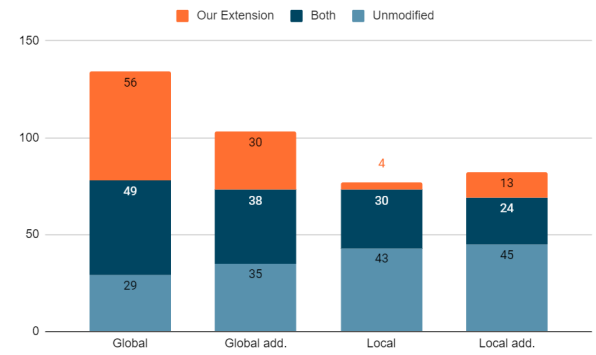

Fig. 1. Evaluation of results. For each setting, we generated 5000 test cases. Each block corresponds to the number of interesting cases found. "Global" corresponds to 4.3, "Global add." to 4.4, "Local" to 4.5 and "Local add" to 4.6.

4.7 Discussion

For each variation our extension discovered new interesting cases. Especially when the conditions are generated by only using global variables our extension works very well. The variations that besides global variables also consider local variables perform significantly worse. This fact is rather unintuitive because we would expect that including more variables into the conditions would result in more complex DCE situations and therefore more interesting cases. One possible explanation could be that the generated dead code is too

complex for the compiler and that no version or option catches the dead code which results in a uninteresting case.

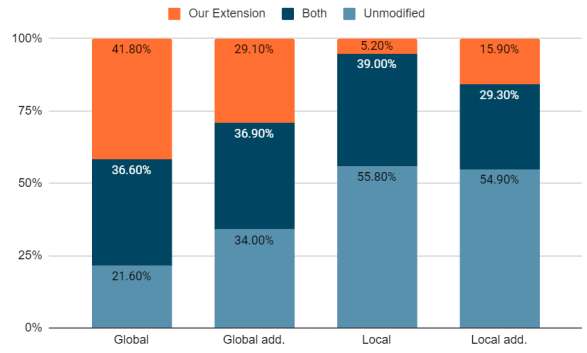

Fig. 2. Distribution of interesting cases.

The idea of adding an additional marker after the unreachable basic block was meant to keep the properties of the original program. If the properties of the original program were maintained the number of cases should increase where both implementations detect interestingness. As can be seen the number of common cases in Figure 1 the result is not successful. Therefore the additional marker does not influence our results positively.

The unmodified programs still discover interesting cases that were not found by our extension. This means the most cases can be found by considering all different modifications together.

In the implementation process we noticed that most variables only reach one value during runtime. This simplifies the optimization process for the compiler. The compiler can consider the variable as a constant and optimize more directly via constant propagation. By modifying `CSmith` to have more variable changes there could be more interesting cases that can be found.

Also considering the variations without additional markers every marker is in a dead basic block. Which means a case would also be interesting if any marker is alive after compilation. By adjusting the interstingness test to our extension it would be possible to catch more intersting cases. This extension goes beyond the scope of this paper.

5 RELATED WORK

Compiler optimizations are as old as compilers themselves. Dataflow Analysis [5] and Dependence Graphs [3] present older methods that allow optimizations. Especially Dataflow Analysis can be used in various ways. For example, Liveness Analysis for register allocation.

Compilers contain many optimization methods that are activated with different compiler options. Sometimes it is also not clear how optimizations will affect the final binary. Even reordering the optimizations can result in a different output. With the high number of factors that influence the result, machine learning methods become essential to get the optimal optimization strategy. In [4] they train a neural network that then helps the compiler to generate efficient loop code. [12] implements the entire optimization process as a machine learning model that tries to find the optimal optimization options for each program to always get the best performance.

Every program depends on the correctness of compilers. For this, there are different methods to check the correctness. The cheapest method is compiler testing [2]. For compiler testing, input programs must be generated. `CSmith` [11] is a tool that generates random C programs. We used the tool to generate the starting point of the inputs. The test oracle problem for compilers is best solved by differential or metamorphic testing. `dead` [8] especially uses differential testing and compares the performance of different compilers.

Another way to show compiler correctness is by formal verification. CompCert [10] is a C compiler that is formally verified with mathematical and machine-checked proofs. It proves the semantic correctness of the resulting binary. The semantic correctness is also important for compiler. To make sure that the optimizations do not change the meaning of the code. Necula [1] introduces a translation validation infrastructure for `gcc` that verifies the semantics after each compiler passes formally.

To be able to remove dead basic blocks compilers need to check the satisfiablility of the path condition. Satisfiability Modulo Theory (SMT) Solvers are programs that solve such formulas. They are widely used for different applications like symbolic execution, program verification and many more. Therefore, testing SMT solvers is essential for the correctness of many programs. Brummayer et al. [7] introduces grammar-based blackbox fuzz testing which generates random inputs that follows the syntax of formulas. With delta-debugging the test cases are reduced to a more manageable size.

Ying-Yang is a more recent project by Winterer et al. [9] that introduces semantic fusion to generate new formulas. For this it takes two formulas and combines them in different ways to get new formulas for testing. Both tools were able to detect many bugs in SMT solvers such as `Z3` and `CVC3`.

6 CONCLUSION

We have extended `dead` with the goal that it detects more performance bugs. The goal is achieved by inserting dead code into randomly generated `CSmith` programs. First `dead` inserts DCEMarkers into every basic block of the program. After this step our extension takes the instrumented program and runs it with `gdb` to get the variable values at the DCEMarkers. With those values we generate unsatisfiable conditions. By wrapping the marker with an if-statement with this unsatisfiable condition we get dead code. To evaluate the performance we compared the number of interesting cases for our extension and the unmodified case. We got the best performance when we only included global variables in our conditions. In this case 41.8% of the intersting cases were found only with our extension. 36.6% cases were found that also were interesting for the unmodified implementation.

A modified interestingness checker is one way to further improve the performance of our extension. Also adding other approaches to add dead code will also find more interesting cases and therefore more performance bugs.

In the end our extension is a first step in making `dead` more effective.

7 APPENDIX

7.1 Result Table

	Global	Global add.	Local	Local add.
Cases	5000	5000	5000	5000
Interesting Cases	134	103	77	82
Both	49 (36.6%)	38 (36.9%)	30 (39%)	24 (29.3%)
Only unmodified	29 (21.6%)	35 (34%)	43 (55.8%)	45 (54.9%)
Only ours	56 (41.8%)	30 (29.1%)	4 (5.2%)	13 (15.9%)

Table 1. Detailed results. "Global" corresponds to the setting in 4.3, "Global add." to 4.4, "Local" to 4.5 and "Local add" to 4.6. "Both" are the number of interesting cases found in both approaches. "Only unmodified" corresponds to the ones only found in dead and "Only ours" only in our approach.

7.2 Source Code

The source code and documentation can be found at https://github.com/carnet98/AST-Project-Carnet-Neffj

7.3 Compiler Versions

Compiler versions used for interestingness analysis.

- gcc-11.2.0
- gcc-10.3.0
- gcc-9.4.0
- gcc-8.5.0
- gcc-7.5.0

REFERENCES

[1] Necula George C. Translation validation for an optimizing compiler. 2000.
[2] Pradel Michael Xiong Yingfei Zhang Hongyu Hao Dan Chen Junjie, Patra Jibesh and Zhang Lu. A survey of compiler testing. 2020.
[3] Padua D. A. Leasure B. Kuck D. J., Kuhn R. H. and M. Wolfe. Dependence graphs and compiler optimizations. 1981.
[4] Wolf Felix Mammadli Rahim, Selakovic Marija and Pradel Michael. Learning to make compiler optimizations more effective. 2021.
[5] A Survey of Dataflow Analysis Techniques. Ken kennedy. 1979.
[6] Cuoq Pascal Eide Eric Ellison Chucky Regehr John, Chen Yang and Yang Xuejun. Test-case reduction for c compiler bugs. 2012.
[7] Brummayer Robert and Biere Armin. Fuzzing and delta-debugging smt solvers. 2009.
[8] Manuel Rigger Theodoros Theodoridis and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. 2022.
[9] Zhang Chengyu Winterer Dominik and Su Zhendong. Validating smt solvers via semantic fusion. 2020.
[10] Daniel Kästner Bernhard Schommer Markus Pister Xavier Leroy, Sandrine Blazy and Christian Ferdinand. Compcert - a formally verified optimizing compiler. 2016.
[11] Eric Eide Xuejun Yang, Yang Chen and John Regehr. Finding and understanding bugs in c compilers. 2011.
[12] Wang Zheng and O'Boyle Michael. Machine learning in compiler optimization. 2018.