# Dead Code Elimination

Robbie Hammond
Case Western Reserve Univerrsity,
School of Engineering
Cleveland, Ohio, USA
reh161@case.edu

Saketh Dendi
Case Western Reserve Univerrsity,
School of Engineering
Cleveland, Ohio, USA
ssd74@case.edu

Milo Cassarino
Case Western Reserve Univerrsity,
School of Engineering
Cleveland, Ohio, USA
mgc73@case.edu

## ABSTRACT

Insert our abstract here.

## KEYWORDS

compiler optimization, dead code

**ACM Reference Format:**
Robbie Hammond, Saketh Dendi, and Milo Cassarino. 2018. Dead Code Elimination. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Dead code, when unaddressed, can be a major problem for a piece of software. If not eliminated by the compiler, dead code can make a program larger, especially if there is a substantial amount of it within a codebase. In addition, dead code can make software arbirarily slower, with computational resources being devoted to declaring unused variables and empty functions. For these reasons, it is imperative that compilers, at a minimum, eliminate chunks of code that are clearly unneeded.

Dead code, as defined in this paper, consists of two related, but distinct types of code. Firstly, code can be considered dead if it is never actually executed at runtime. For example, code inside of the 'else' portion of an if-else block where the condition is always true would be dead. Secondly, a portion of code can be considered dead if the computation performed on those lines are never used anywhere else. For example, a function that is never written to or read from can be considered dead.

Our project focuses on adding basic dead code elimination strategies into the compiler created in the fourth class project. Our goal is to give the compiler support for eliminating unreachable code in if branches, for loops, and while loops. In addition, the compiler should be able to remove unused variables and other miscellaneous dead code optimizations.

## 2 IMPLEMENTATION

In this section, we will outline how we implemented the aforementioned code removal functionality.


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

### 2.1 Class and Function Additions

In order to implement our optimization pass, a number of additional functions and classes had to be created. This section will give an outline of the important modifications, omitting many details and less important additions.

*2.1.1 AST.cpp and function.cpp Changes.* The creation of the Optimize() function is the most important change to the AST class. The Optimize() function simple calls PerformOptimization() on each function. The PerformOptimization() function calls CanOptimize(), which is a function that both optimizes a statement and returns it's ability to be optimized. This function is described in more detail in the subsequent sections.

*2.1.2 Statement.h Changes.* Within the Statement.h file, a number of additional functions have been declared, with the most important being CanOptimize() and HowToOptimize(). The purpose of these functions are relatively self-explanatory. CanOptimize() returns if a statement is able to be optimized. For most types of statements, we do not try to optimize them, so the function returns false by default. HowToOptimize() tells the rest of the program what must be done to optimize the statement. HowToOptimize() returns an *Optimization*, which is an enum with values such as *REMOVE_LOOP*, *REMOVE_ELSE*, *NO_OPTIM*, and others.

*2.1.3 Block.cpp Changes.* The CanOptimize() function within Block.cpp is where much of the important optimization work is done. As mentioned above, CanOptimize() is defined with Statement.h and child classes inherit and can override this function. Though all other subclasses of statement simply return a true or false value from the function, the CanOptimize() function in Block.cpp will modify the state of the containing block class to reflect the optimization changes. Generally speaking, these state changes are the creation and/or deletion of statements within the statements list.

### 2.2 If Statements

There are two cases in which a branch of an if statement will never run. The first case is when the condition is always true, regardless of the state of the program when the if statement is entered. The second case is when the condition is always false.

*2.2.1 Condition Always True.* To check if the condition is always true, we get the compiled R value of the condition. If it is 1, then we know that the condition must always be true. When this happens, we don't need to have any kind of branching logic in the AST. All we must do is generate the code within the else statement and generate the rest of the program like normal.

*2.2.2 Condition Always True.* To check if the condition is always false, we get the compiled R value of the condition as in the other

case. If it is 0, then we know that the condition must always be false. Similarly to the case when the condition is always true, all we must do is generate the code within the then statement with no branching logic.

## 2.3 Loops (For And While)

Just like with the if statements, there are two cases in which a loop may cause dead code. The first case is when the condition is always true, and the second case is when the condition is always false.

*2.3.1 Condition Always True.* Just like with if statements, we can check if the condition is always true by checking the compiled R value of the condition. If it is, then we know that the loop will never terminate, rendering the subsequent code unreachable. To simplify the AST, we can simply get rid of all subsequent statements after the loop. For consistency sake, a return statement is still appended at the end of the loop, with the return type matching that of the function. Since the code is unreachable, the value of the return is not important, and arbitrary simple ones were chosen.

*2.3.2 Condition Always False.* If the compiled R value of the condition is always false, there is no need to include the loop at all within the AST or LLVM IR. As such, the entire loop is erased within the AST, with the rest of the code being left unaffected.

## 2.4 Useless Variables

Hopefully we can get this implemented and talk about it. Worst case, we can just talk about things we considered and stuff we tried.

## 2.5 Other Optimizations

An additional simple dead code elimination technique is to eliminate adjacent statements. For example, if the statement $x = 1$ is directly followed by $x = 1$, one of these statements can be removed with no effect on the code.

# 3 MAJOR LIMITATIONS

There still is a large number of limitations with our optimization pass. This section will outline some of the most critical ones.

*3.0.1 Return/break statements within infinite loops.* An infinite loop is not really infinite if there exists a reachable jumping statement within it. Our optimization is not able to recognize this, so as long as the condition is always true, it assumes the following code is unreachable.

*3.0.2 Resolving more complicated conditions (specifically in regard to the AST).* Our optimization pass can only recognize conditions as always being true or false when the condition is either a '0' or '1'. Obviously, this is very much limiting, as most conditions that are always true/false aren't so obvious. However, it is worth noting that code has been written that allows IR to not be generated for slightly less trivial conditions, such as when a variable is assigned to 0 and then used in the condition. Since the optimization pass is only supposed to operate on the AST, this logic has been commented out. See for.cpp, while.cpp, and if.cpp for how this is done.

# 4 TESTING METHODOLOGY

To test our optimization pass, a number of simple tests were created. The main metric we decided to use to determine the effectness of our optimization pass was size of the generated LLVM IR, in bytes. For many of our tests, we also recorded execution time. This metric was omitted for tests containing infinite loops, as those programs do not halt.

# 5 RESULTS

We'll use tables, charts, and other stuff to make our basic testing look fancy. The better it looks, the more it'll look like we tried.

# 6 CONCLUSION

We can talk about our results and talk about how the simple examples could extend to bigger examples and that big code bases need dead code elim. Blah blah blah.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

**Table 1: File Size Test Results**

| File | Unoptimized IR Size | Optimized IR Size |
|---|---|---|
| *if_remove_then.c* | 848 | 348 |
| *if_remove_else.c* | 847 | 348 |
| *infinite_for.c* | 905 | 716 |
| *useless_for.c* | 936 | 321 |
| *infinite_while.c* | 888 | 699 |
| *useless_while.c* | 833 | 321 |

**Table 2: File Execution Time Test Results**

| File | Unoptimized IR Size | Optimized IR Size |
|---|---|---|
| `\author` | 100 | Author |
| `\table` | 300 | For tables |
| `\table*` | 400 | For wider tables |