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, 2 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.

2 IMPLEMENTATION

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

2.1 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.

2.1.1 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 generate any kind of branch logic in the LLVM IR. All we must do is generate the code within the else statement and generate the rest of the program like normal. Refer to the first portion of the Compile function in if.cpp to see percisely how this was implemented.

2.1.2 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. Again, refer to the first portion of the Compile function in if.cpp to see percisely how this was implemented.

2.2 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.

2.2.1 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 shorten the

2.3 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.

3 TESTING METHODOLOGY

I'm thinking our main metric should be file size in bytes, since this is the main purpose of dead code elimination. We can also measure speed as well, but make it clear that it isn't as significant.

4 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.

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

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

Table 1: Some Typical Commands

Command	A Number	Comments
`\author`	100	Author
`\table`	300	For tables
`\table*`	400	For wider tables

5 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