Dead Code Elimination on Abstract Syntax Tree Intermediate Representation

Ceci Cohen
clc169@case.edu
Case Western Reserve University
Cleveland, Ohio, USA

Christopher Danner
cld99@case.edu
Case Western Reserve University
Cleveland, Ohio, USA

Joey Li
xxl1021@case.edu
Case Western Reserve University
Cleveland, Ohio, USA

ABSTRACT

Code optimization is a fundamental compiler technique that enhances the space and runtime efficiency of code upon compilation. LLVM includes many optimization passes, such as dead code elimination, which are designed to operate on LLVM IR code. In this report, we present our own implementation of the dead code elimination (DCE) pass that operates at the abstract syntax tree (AST) level and compare its performance against the LLVM execution of the same pass and the unoptimized code. The optimization passes were performed with respect to execution time and code. Our evaluation revealed that space and time efficiency increased relative to non-optimized code after performing the optimization pass. Our pass produced a demonstrable decrease in compiled code size and execution time over unoptimized code, with an average of 11.1% decrease in compiled code size across all functioning tests and an average of 49.5% decrease in execution time across all performance tests.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies** → *Optimization algorithms.*

KEYWORDS

Compilers, Compiler Optimization, Dead Code Elimination, Abstract Syntax Trees

ACM Reference Format:
Ceci Cohen, Christopher Danner, and Joey Li. 2024. Dead Code Elimination on Abstract Syntax Tree Intermediate Representation. In . ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the field of code optimization, code optimization can be divided into machine-dependent or and machine-independent optimization. Machine-dependent optimizations are performed in the later stages of the compilation process and act directly on machine code, while machine-independent optimizations act on the intermediate representation that is generated in the intermediate stage of compilation. Dead code optimization is an example of machine-independent optimization, operating on an intermediate representation such as three-address code or an AST to eliminate dead code segments. In this context, dead code refers to segments of code that have no effect on the output of a program due to being unreachable, redundant, or otherwise unnecessary. This includes dead variables (variables which are reassigned a new value before being used or are never used), dead programming structures (functions and types which are declared but never used), and unreachable code (code following returns, breaks, or other unconditional jumps; code in always-false conditionals; commented-out code).

The elimination of such code can improve the performance of compiled programs by eliminating unnecessary computations to reduce program execution time and overall code size. Our proposed implementation operates on an AST, which is a commonly utilized form of intermediate code representation which was utilized in Programming Project 4 but is unsupported by the existing LLVM DCE pass. We implement some of the primary instances of dead code elimination, including dead variable elimination and unreachable code elimination, with the specific scope of our implementation discussed in greater detail in the next section.

Conference'17, July 2017, Washington, DC, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN XXX-X-XXXX-XXXX-X/24/05
https://doi.org/XXXXXXX.XXXXXXX

2 IMPLEMENTATION

Our dead code elimination pass is implemented through a modification to the base compiler created in Programming Project 4. The function `DeadCodeEliminationPass`, runs all individual dead code optimization passes and is included in the source code file `ast.cpp`, which defines the representation of the abstract syntax tree in the compiler. The two optimization passes, `EliminateUnreachableCode` and `EliminateDeadCode`, eliminate unreachable code in always-true or always-false conditional blocks and dead variables respectively. The specifics of these implementations are discussed below.

2.1 Identifying Dead Code

2.1.1 Dead Variable Elimination. Analogously to how a DCE pass on IR code performs a backward traversal of the IR code, `EliminateDeadCode` performs a backward traversal of the AST relative to the execution order. That is, it performs a right-to-left postorder traversal of the AST, exploring the child nodes from right to left before operating on the current node. This traversal structure was implemented with a recursive function.

To determine which variable assignments can be removed, a map of the variable IDs and respective statuses is maintained throughout the traversal. When the traversal reaches a variable use (any variable reference which is not the left-hand side of an assignment), it updates the respective variable to live in the map, adding the variable if not already included in the map. When the traversal reaches a variable assignment, it checks the map and removes the assignment if the variable is not live or is not in the map. It then resets the variable status to dead in the map to show that the variable has been reassigned (or was not used in the first place).

Special attention is paid to more complex dataflows for if statements, for loops, and while loops. For if statements, the state of the variable map at the end of the if statement (entering the body) is copied into a second map. Each map is used to track variable reassignments and uses along one of the two dataflow paths (either then or else/none), then the maps are combined once the front of the if statement is reached. Any variable which was live along either dataflow path is marked as live in the combined map. For loops and while loops operate similarly, however additional consideration is made for the looping structure. Prior to any elimination of dead variables, a pass is made over the entire looped body of the conditional, body, and increment (for loop only) to obtain the live status of variables at the beginning of the looping structure. This is then combined with the variable state table at the end of the loop in the same way that the branching paths for the if statement are handled. The pass continues through the loop before the resulting table is combined once again with the table at the end of the for loop, to account for the possibility that the conditional is never satisfied.

While our pass is implemented to eliminate dead variable assignments throughout most of the AST, there are some limitations. Our pass only makes a single traversal of the AST, so it will consequently miss variables which are only live due to the use in a dead variable assignment. Additionally, due to some limitations in my PG4 implementation of the for loop, the full elimination of a particular field in the for loop header could lead to compilation errors. Despite these limitations, our pass nevertheless covers many of the common instances of dead variables.

2.1.2 Unreachable Code Elimination. Unreachable code elimination is implemented in the same traversal as dead variable elimination, however it operates as a forwards (preorder) traversal of the AST because `EliminateUnreachableCode` is called immediately upon discovery of a relevant node. The pass examines the control-flow blocks `if`, `for`, and `while` by identifying always-true or always-false conditions. `EvaluateExpression` determines the value of the conditional expression, which `EliminateUnreachableCode` uses to determine whether any code is able to be eliminated.

`EvaluateExpression` determines the value of a conditional by recursing until a definite `true` or `false` expression is reached, and then calculating the value of each boolean expression. Three possible return values exist - 0, 1, and 2 - which indicate an always-false, always-true, and uncertain expression respectively. The type of expression determines how the function output is handled. When `EvaluateExpression` returns always-false, code within `for`, `while` and simple `if` statements will never be reached. When `EvaluateExpression` returns always-true, code within the `else` block of `if...else` statements will never be reached.

This pass was unable to be effectively implemented due to some limitations of the AST. During parsing, boolean expressions are first converted to integer values, then cast back to boolean values. Because `EvaluateExpression` was only implemented for boolean values and operations on boolean values, this means that the function cannot successfully predetermine the value of loop conditional to identify unreachable code to eliminate.

2.2 Removing Dead Code

The dead code elimination pass exclusively removes nodes from the AST. For dead variable elimination, the pass removes assignments to variables that are not used between the assignment in question and the next assignment to that variable (or the end of the current scope in which the variable was declared, whichever is earlier). For unreachable code elimination, the pass removes all nodes contained within conditional statements or loops which the program execution will never reach.

2.3 Maintaining Correctness

Since the pass exclusively eliminates code that is never executed or whose execution does not affect the output of the program, it should theoretically preserve code correctness without issue. In practice, the pass needs to ensure that all other essential AST links are preserved and that code is not marked as dead and eliminated when it is actually needed elsewhere in the program.

In order to ensure that all essential AST links are preserved, we need to take special care with assignments whose return values are used by another expression. For dead variable assignments whose returns are used, the assignment is replaced in the parent node with its right-hand side. This preserves the value that would have been returned by the assignment, allowing the assignment to be bypassed. For standalone statements such as unreachable statements and dead variable assignments whose return values are unused, the statements can simply be removed from their parent node without changing the program execution. We also tried to preserve the right-hand sides of dead assignments with function calls or other assignments on their left-hand sides similarly to dead assignments whose return values are used. Due to the limitations of our choice of pass structure, however, we were only able to preserve function calls and assignments which are directly on the right-hand side of the dead assignment, not function calls and assignments which are nested within other expressions such as OR and AND statements.

In order to ensure that code is not incorrectly marked as dead, we observe two conditions for dead variable elimination and unreachable code elimination respectively. For dead variable elimination, we ensure that all viable paths through a conditional statement are considered by marking a variable as live going into the conditional statement if it is live in any dataflow path through the conditional. We also ensure that for variables located within a loop, they are marked as live if they are live at the beginning or end of the main loop structure (conditional, body, and increment, if any). The process used to achieve this dataflow analysis is covered above when discussing the implementation of the pass. For unreachable code elimination, we only eliminate the bodies of conditional statements and loops which we can be certain will not be executed. This is done by evaluating the condition of the conditional statement or loop, and only marking code as unreachable if we can obtain an always-true or always-false value. If there is any uncertainty, we leave the code untouched.

3 TESTING METHODOLOGY

We implemented five test cases, named `test1.c` through `test5.c`, to evaluate the correctness of various aspects of the optimization

Table 1: Compiled LLVM IR Code Sizes (in Bytes)

Opt. Level	performance1	performance2
Unoptimized	7859	3561
AST DCE	7177	3118
LLVM DCE	7625	3411

pass. These tests are located within the `tests` folder of the project, along with two larger test programs named `performance1.c` and `performance2.c` used to evaluate performance. The tests are described in greater detail in the `README.md` file of the project, however below is a short overview of what each program tests:

(1) `test1.c`: Tested variable uses in all major classes of nodes without dataflow complexities (i.e., binary expressions, unary expressions, function calls, assignment statements, and return statements)
(2) `test2.c`: Tested dead variables in all major classes of nodes without dataflow complexities (i.e., statement blocks, binary expressions, unary expressions, function calls, assignment statements, and return statements)
(3) `test3.c`: Tested variables dead through varying numbers of dataflow paths through if for and while statements, and dead variables within such statements
(4) `test4.c`: Tested unreachable code in if, for, and while statements
(5) `test5.c`: Tested corner cases such as dead assignments with function calls or live assignments as their right-hand sides

All of the tests executed successfully with expected results, with the exception of `test4.c`. As previously mentioned, the `EvaluateExpression` was unable to successfully determine the values of the conditional expressions.

The larger test files, `performance1.c` and `performance2.c`, were used to evaluate the optimized code based on two metrics: execution time and code size. Code size could be evaluated trivially by checking the size of the LLVM IR code files. Since the existing LLVM DCE pass operates on the LLVM IR code, the size of the LLVM IR code was the preferred measure of code size compared to the size of the AST since it allowed us to compare our pass to the existing pass as well as the unoptimized code. Execution time was measured using `-time-passes` over 50 executions of the LLVM IR code with `lli`. Internal timing was not used due to the additional overhead of needing to include C++ timing code when performing timing internally, which would complicate analysis of the IR code. The results of this evaluation are discussed below.

4 RESULTS

As mentioned above, code size was evaluated by checking the sizes of the compiled LLVM IR code files. The results across all test files are reported in Tables 1 and 2.

As Tables 1 and 2 show, our AST DCE pass exhibited consistently lower IR code sizes compared to the unoptimized code, with the exception of `test4`, in which the optimization pass was unsuccessful. The LLVM DCE pass also exhibited reduced code size, however it was surpassed by our DCE pass in some cases. This may have been due to dependencies on other optimization passes to obtain maximal effectiveness or the fact that our optimization pass is more directly focused on the chosen IR (an AST).

Table 2: Compiled LLVM IR Code Sizes (in Bytes)

Opt. Level	test1	test2	test3	test4	test5
Unoptimized	1763	1716	6611	9496	1207
AST DCE	1633	1270	6383	9496	1101
LLVM DCE	1528	1717	-	-	1069

Table 3: Avg. Runtime of LLVM IR Code (in Seconds)

Opt. Level	performance1	performance2
Unoptimized	10.83	2.97
AST DCE	4.59	1.74
LLVM DCE	8.15	2.72

Execution time was evaluated using `-time-passes` with the `lli` command. The average execution times the compiled LLVM IR code across 50 iterations is reported in Table 3.

As Table 3 shows, our AST DCE pass exhibited substantially lower average execution time compared to the unoptimized code. The LLVM DCE pass also exhibited reduced execution time, however it was surpassed by our DCE pass in some cases. Once again, this may have been due to dependencies on other optimization passes to obtain maximal effectiveness or the fact that our optimization pass is more directly focused on the chosen IR (an AST).

Overall, our AST DCE pass proved to be effective in eliminating dead variable assignments, reducing compiled code size and execution size and removing all expected variable assignments covered by our implementation of the pass. The pass was less successful in eliminating unreachable code, failing to achieve significant optimization. Between the unsuccessful unreachable code elimination and other instances of dead code which were not implemented (such as dead function elimination and more advanced dead variable elimination), we still have significant room for potential improvement, however we made significant progress in improving over unoptimized code.

5 CONCLUSION

In this study, we presented an implementation of a dead code elimination pass that operates at the AST level. Our AST-based DCE pass performs two key optimizations: eliminating dead variables and unreachable code blocks. The dead variable pass utilizes a backward traversal of the AST, identifying and removing assignments to variables that are not utilized in subsequent code segments. Through a recursive postorder traversal, we effectively identified dead assignments and removed them from the AST, optimizing the code without compromising its functionality. Our unreachable code elimination pass utilized a forward traversal of the AST to identify and eliminate unreachable segments, however it encountered difficulties with accurately predetermining condition values and was unable to operate with full effectiveness.

We evaluated the performance of our optimization pass by testing the size of the compiled program and found that there was a net decrease in the average size and runtime of the compiled optimized programs, comparable to and in some cases better than the existing LLVM DCE pass. These results demonstrate that performing dead code elimination on the higher-level AST representation can yield better performance relative to unoptimized machine code, and a tailored optimization pass for your particular IR implementation may have better performance than generally-available ones. This AST-level DCE can serve as a useful optimization phase, potentially complementing existing IR code-level optimization passes. Future work can explore combining this AST-level pass with other optimizations and evaluating its performance across different computer architectures.

Received 4 May 2024