Compiler Optimization: Dead Code Elimination Parallel Computational Tree Logic

Jessy Liao
Department of Computer Science
Colorado School of Mines
Golden, Colorado
jessyliao@mymail.mines.edu

***Abstract*—Dead code elimination is a compiler optimization in which code that doesn't affect the overall function of a program can be removed. Eliminating dead code can lead to benefits such as avoiding unnecessary computations, smaller program size, and a more simplified program structure.**

This work explores using computational tree logic (CTL) as a solution to dead code elimination and how viable the technique is compared to the Mark Sweep method, a commonly used method for detecting dead code. There are several aspects that are considered when determining how viable a CTL algorithm is, this includes the different type of dead code it catches, how fast the algorithm runs, and how correct it is in detecting dead code.

Through the tests conducted, the CTL algorithm is able to detect more unique cases of dead code, however, it sacrifices computational time in doing so. Through integrating parallelization on the CTL algorithm, the computational time required is decreased drastically, and is comparable to the Mark Sweep algorithm.

I. Introduction

Although dead code seems harmless due to its nature of not affecting the behaviour of a program, there are indirect issues that it could cause. Code produced by large software companies are passed off from developer to developer. If dead code is never addressed, useless code will be perpetuated through multiple versions of the program. This leads to code that the next developer needs to waste time understanding, but never using.

There are two main types of dead code that exist, unused code and unreachable code. Unused code are lines of code that are never utilized. An example of this is variables being initialized and declared, but never used in any operations or calls. Unreachable code is code that is never reached in the program. An example of this is code that exists after a return statement is called. The return statement would stop the function and all code after it is unreachable. The example provided above are trivial. There are multiple cases that can lead to the problem of dead code compounding and becoming much more difficult to detect. For example, if a variable is only used in an if statement, and later on it is discovered that the if statement is unreachable, that would lead to the variable in that if statement becoming dead. The problem of detecting dead code becomes much more difficult when analyzing more complex programs. Figure 1 provides an example of how difficult dead code detection can become.

```
1 int g = 3
2 int c = 3
3 while ( g < 10 )
4       g = g + 1
5       if ( g == 11 )
6               c = 8
7               print(c)
8  return g
```

Fig. 1. Initially, c seems like a used variable because it's used in the print function (line 6 and 7). However, the if statement (line 5) is actually always false, and therefore the code inside of it is unreachable. This makes c (line 2) a dead variable

Most dead code elimination techniques rely on iterating over all the possible states the program could be, and determining whether a line of code is ever used. Both the CTL and Mark Sweep algorithm uses this technique in order to detect dead code.

II. Background

A. Abstract Syntax Tree

In most compilers, an abstract syntax tree (AST) is usually generated. An AST is a tree that represents a program. The tree is an abstract representation of the code used in the program. Each node in the tree represents the different states the program goes through. Each node points to the next state the program can transition to. Traversing the tree with depth-first search gives you the order of the program execution. Loops do not create cycles in the tree so when implementing the dead code elimination algorithms, a creative solution had to be used to iterate through the loops.

Since the AST contains enough information to iterate through the entire program, but also remains abstract enough where it doesn't take up a lot of space, it is a perfect data structure to apply the dead code elimination algorithm. For this dead code elimination implementation, it was assumed that the user would input an AST, which the algorithms would use to detect which nodes are dead.

B. Mark Sweep Review

The Mark Sweep algorithm is a technique used to detect dead code, it is a naive algorithm and works in two steps.

The first step is called the mark phase in which the algorithm iterates through the different states of a program. For each state the algorithm successfully reaches, its marked. The second step is called the sweep phase where it goes through all the states in the program and checks whether they are marked or not.

This technique is a trivial solution, and doesn't actually catch all cases of dead code. Unreachable code is not detected in this algorithm. If the code is not reachable, only the variables will be picked up in the sweep phase. Everything else will be considered as marked still.

For the purpose of this study, the algorithm doesn't need to be fully complete. It is able to detect unused variables, and is relatively quick. Therefore it will be utilized as a benchmark to compare the CTL implementation against.

C. Computational Tree Logic Review

A CTL is a branching time logic that can be used in model-checking algorithms. A model-checking algorithm is an algorithm where a specification or formula can be applied to a finite-state model, and the algorithm can determine whether the model fits the specification or formula.

CTL gives us a grammar in which we can use to construct formulas, or specifications, to be used in a model-checking algorithm. The following grammar below is the CTL syntax grammar:

$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \mid$ AX ϕ | EX ϕ | AF ϕ | EF ϕ | AG ϕ | EG ϕ | A $[\phi$ U $\phi]$ | E $[\phi$ U $\phi]$

The following defines the operators used in the grammar above:

- A ϕ = Along all paths, ϕ has to hold
- E ϕ = Along at least one path, ϕ holds
- X ϕ = Next (has to hold at the next state)
- G ϕ = Hold for the rest of the path
- F ϕ = Eventually has to hold
- ϕ U ψ = ϕ has to hold until ψ holds
- ϕ W ψ = ϕ has to hold until ψ holds (ψ doesn't have to hold)

With the given syntax above, we can construct formulas such as AG (variable $\implies$ AF used) which states that given any state of a program, if a variable exists, it will eventually be used. With a sufficient model to represent a program, we can apply formulas such as the one described above to determine whether the model meets the specification of containing no dead code.

This idea is used to implement the CTL dead code elimination algorithm. The model that represents the program will be the AST that a user inputs. Within the algorithm are a set of formulas in which outlines the state a program can be in without it breaking the no dead code specification.

III. Implementation

CORONER (coroners investigate dead people, CORONER.py investigates dead code) is the name of model-checker program that was created, and which I used to explore how viable it is to use CTL as a dead code elimination algorithm. Before implementing the CTL and Mark Sweep algorithm, a model had to be created for the algorithms to search through. CORONER takes in a text file that represents the different nodes in a AST, and what other nodes it points to. From that input file a tree structure is created with all the necessary information for the model to run the two algorithms. Each node in the AST contains the following information:

- ID: to differentiate from node of the same type
- NAME: name of the node (These are name of variables)
- TYPE: type of the node (Variable, If, While, etc...)
- CHILDREN: list of children nodes
- MARK: used in Mark Sweep algorithm

With the tree structure fully defined and populated with data from the input text file, the AST can now be passed to the two different algorithms. Due to the scalability of the problem (there are a lot of different cases of dead code to check for) CORONER only considers the following four cases of dead code:

- Unused variables
- Unused parameters
- Unreachable code due to if statements evaluating to always true or false
- Unreachable code due to code being written after a return statement

A. Mark Sweep Implementation

The Mark Sweep implementation takes in the AST and iterates through each of the node in the tree. It marks each node that it encounters. For the Mark Sweep algorithm to properly work, a helper function had to be created to loop sub-trees that represented loops. This helper function takes in the sub-tree that represents a loop, and it calculates the conditional and runs the loop while the conditional is true. In order to calculate the conditionals, the Mark Sweep algorithm has a dictionary that stores all the variables used in the program, and what their current value is. If the algorithm encounters a return node (return statement), it immediately breaks out of the algorithm.

The second phase was easy to implement. All it had to do was iterate through the AST and check for all nodes that weren't mark. The remaining nodes are dead variables. Like previously mentioned, this algorithm isn't able to detect unreachable code. Even though there is a check to stop marking after encountering a return statement, the algorithm actually considers only the variables after the return statement as unmarked. The algorithm doesn't say anything about code that isn't variables and whether they are reachable or not. In terms of speed, the algorithm only does two sweeps through the AST. We will see that this is relatively quick when compared to the CTL implementation.

B. CTL Implementation

For the CTL implementation the algorithm has four different functions that it calls. The four functions relate to the four different cases of dead code.

The function for unused variables kept track of all the ”used” variables it encounters. Used variables are variables that are called by a function or are in return statements. The function also keeps track of all the variables that are referenced by the used variables. For example if you had a used variable a that was returned, and a second variable b was used to calculate a, then b is an alias of a and is also considered a used variable. When the function finishes iterating through the AST it will terminate with a list of all used variables. At that point any variable not in used is a dead variable.

The function for unused parameters works exactly the same as the unused variable function with a minor tweak where it keeps track of parameters instead.

Both the unused parameters and variables function are fast because it doesn't need to integrate the helper function that loops through the while loop.

For the unreachable code due to if statements function, the code is similar to the Mark Sweep algorithm. It has to iterate through the entire AST. If it encounters a loop it has to iterate the sub-tree until the loop ends. While iterating through all the node, the function keeps track of all the if statements that it has encountered and all the boolean values the conditional statement produced at that state. When the function finishes iteration, if an if statement evaluated to the same boolean value every iteration, it has dead code.

The unreachable code due to return statements is similar to the unreachable code due to if statement function. It also iterates through the entire AST, and accounts for loops. If at any point it encounters a return statement node, it will start adding all nodes after that into a list. This list is all the unreachable code.

IV. Evaluation: Part I

After running multiple experiments on AST, the Mark Sweep algorithm is always faster than the CTL algorithm. This is because in the Mark Sweep algorithm it only iterates through the AST once, with loop consideration, and then once more time without loop consideration. The CTL algorithm runs through the AST twice, with loop consideration, and twice without.

Although the Mark Sweep algorithm is about twice as fast as the CTL algorithm, the CTL algorithm produces a much more correct result. The CTL algorithm is able to detect unreachable code which the Mark Sweep algorithm cannot.

V. Parallelization

Due to how the CTL algorithm is structured, parallelization techniques can be implemented easily to obtain more speedup from the algorithm. The CTL algorithm is split up into four different formulas which can all be executed in parallel. Using the multiprocessor library in python. With the library I created four different processes, one for each of the different formulas, and then executed them in parallel. Each function reads from the AST which is never changed, and therefore there are no memory conflicts. Each process simply prints out all the dead code it discovered at the end of its execution.

VI. Evaluation: Part II

Now with the added parallelization the CTL algorithm essentially takes about the same amount of time as the Mark Sweep algorithm. This is because the two AST sweeps, with consideration of loops, now can be done in parallel. This is the same with the two AST sweeps without consideration of loops.

VII. Conclusion

Although CORONER checked for four cases of dead code, this problem can easily be extended to check for other cases without sacrificing computational time. More formulas can be created for the CTL algorithm to handle different cases of dead code. Due to the ease of implementing parallelization into CTL, adding more formulas won't affect computational time. The only case in which computational time is affected is when you run out of cores or threads to parallelize on. However, if this does occur, this program can easily be extended to GPU parallelization which would give you a significant amount of more cores depending on your hardware.

The only case in which CTL algorithm could fail that I discovered was the cases of infinite while loops. CTL model-checking can only produce a result if the algorithm is able to terminate. In the case of an infinite while loop, the algorithm won't terminate, and thus won't produce a result. Dead code optimizer today don't usually detect infinite while loops, usually they time out. Therefore this would mean CTL algorithm can be made to handle as many cases as actual dead code optimizer.