Final Report
Advanced Compiler Construction

2017-20845 Wonjae Jang
2018-25193 Dongwan Kang

December 18, 2018

1 Introduction

For three months after we have submitted the project proposal, we completed the implementation of Constant propagation & Dead code elimination and Function Inlining. From this project, we have not only been able to learn compiler optimizing techniques, but also have a better understanding of compiler.

In this document, we describe how we implemented Constant propagation & Dead code elimination and Function Inlining. Those include data structure, implementing method (technique & algorithm), the result of code optimization.

2 Optimizing compiler

In computing, an optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program. Our focus is to minimize the time taken to execute a program and the size of the execution program. There are various scopes of optimization techniques. Among them we implement Constant propagation & Dead code elimination and Function Inlining.

All optimization was performed on the low-level IR(three address code) of SnuPL/1 compiler given. We made three classes (ConstantP, DeadCodeE, Inlining)to perform each optimization technique. And we made some data structures(BasicBlock, VariableDef, fNode, fCallNode). Each class's main method receives a CModule object (TAC) of an entire program as parameter and performs its work. In working flow of compiler, these works are performed after generating TAC code and before generating assembly.

We optimizied the compiler with following steps:

1. Perform constant propagation and dead code elimination.
2. Inline some functions.
3. Perform constant propagation and dead code elimination.

We did constant propagation and dead code elimination before function inlining. This is because original code could have unnecessary code and this could affect inlining order. Then we inlined some functions. Last, we performed constant propagation and dead code elimination again because function inlining could make unnecessary codes.

3 Constant propagation & Dead code elimination

In compiler theory, dead code elimination (also known as DCE) is a compiler optimization to remove code which does not affect the program results. Constant propagation is the process of substituting the values of known constants in expressions. So, constant propagation is also a method to remove code which does not affect the program results. Removing such code has several benefits: it shrinks program size, an important consideration in some contexts, and it allows the running program to avoid executing irrelevant operations, which reduces its running time. It can also enable further optimizations by simplifying program structure. Dead code includes code that can never be executed (unreachable code), and code that only affects dead variables (written to, but never read again), that is, irrelevant to the program.

3.1 Basic Code Block

To implement Constant_Propagation & Deadcode_Elimination, we partition the intermediate code(.tac) into basic blocks, which are maximal sequences of consecutive three-address instructions.

Basically, basic blocks have the properties that

1) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

2) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

In snuplc's intermediate code, the target of a conditional or unconditional jump is a jump label. So, we can begin a new basic block with the first instruction or the lable and keep adding instructions until we meet another label on the following instruction.

```
class BasicBlock {
    public:
        ...

    protected:
        CTacInstr* _header;
        int _header_id;
        list<CTacInstr*> Instr_list;
        list<BasicBlock*> _predecessor;
};
```

Listing 1: Data structure for BasicBlock

This data structure has just information of predecessors, not successors , because we deal with forward data-flow analysis in our project.

3.2 Constant Propagation

Constant propagation is the process of substituting the values of known constants in expressions at compile time. And Constant folding is the process of recognizing and

evaluating constant expressions at compile time. We implemented these two methods together and will call them as Constant propagation. Constant propagation is implemented using reaching definition analysis results. If all a variable's reaching definitions are the same assignment which assigns a same constant to the variable, then the variable has a constant value and can be replaced with the constant.

As described in the text book, the reaching definitions problems is defined by the following equations:

$$OUT[B] = gen_B \cup (IN[B] - kill_B) \tag{1}$$

$$IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P] \tag{2}$$

We can make an array indicating reaching definition at a program point of each block with the equations. If a variable is constant and have sole variable name, we can decide the variable as liveConstantDefinition, which can be replaced with constant value.

The algorithm is sketched by the following :

Algorithm 1: Calculating liveDefinition in a Block

```
Data: BasicblockList
Result: liveConstantDefinition
liveConstantDefinition = {};
while changes to any OUT occur do
    forall B ∈ BasicblockList do
        OUT[B] = gen_B ∪ (IN[B] - kill_B)
        IN[B] = ∪_{P a predecessor of B} OUT[P]
    end
end
forall B ∈ BasicblockList do
    liveConstantDefinition = IN[B] ∩ OUT[B] - Def_{overlapedName}
end
```

The actual implementation of Constant Propagation consists of the following steps.

1) Partition the code into basic blocks.

2) In block, remove a redundant assignment and replace a constant variable with a constant.

3) Calculate an unary operation of a constant and replace it as a new assignment.

4) Make reaching definition analysis and decide a constant variable in each block.

5) Replace a constant variable with a constant.

6) Calculate an unary operation of constants and replace it as a new assignment.

7) Calculate an relation operation of constants in the conditional branching, and replace it with unconditional branching.

There are two constant replacement steps, step 2) and step 6). First one is an local processing in a block. If there is an constant assignment of a variable in a block, we can replace all the variables used after the assignment with the constant in the block. Second one is an inter-processing between blocks in a function. This replacement can be processed after analysis of variable usages using reaching definition.

And result of a step needs to be processed again in the previous step. For example, we can get a new constant assignment through step 6), this assignment should be processed at the scratch. So, we repeat the entire processes until no further changes occur.

3.3 Dead code elimination

Dead code is normally considered dead unconditionally. Therefore, it is reasonable attempting to remove dead code through dead code elimination at compile time. There are several techniques of dead code elimination. Among them, we implemented to remove unused variable definitions and remove unreaching code blocks and unreaching instructions by unconditional branching.

First, we can find unused variable definitions after scanning operands of all operations. And we can find unreaching code blocks scanning destination label of branching operations except for the following code block of non-branching code block. And we can find unreaching instructions scanning the following instructions of unconditional branching instruction.

Below algorithm is to eliminate unreaching instructions in code block by unconditional branching.

Algorithm 2: Removing eliminate unreaching instructions

```
Data: BasicblockList
forall B ∈ BasicblockList do
    removeFlag = false
    forall Instr ∈ InstructionList[B] do
        if remove == true then
            DelInstr(Instr)
        end
        if Instr == opGoto then
            removeFlag = true
        end
    end
end
```

4 Function Inlining

We implemented function inlining. First, we parsed the original TAC file. Then, we scored each function call. Pick the highest scored function call and inline it. Next, we updated the parsed data to repeat picking and inlining. Last, we repeat from picking a target to updating parsed data until it reaches termination condition.

4.1 Data Structure

Data structure of function inlining consists of two classes: fNode and fCallNode. fNode represents every declared function, and it contains fCallNode which represents function call in every fNode. Variables in each function is shown is Listing 2.

```
class fNode {
  public:
      ...
```

```
  protected:
  int _code_length;
  vector<fCallNode*> _children;
  string _name;
  CScope* _module;
};

class fCallNode {
  public:
      ...

  protected:
  float _score;
  int _loop_level;
  fNode* _node;
  string _name;

};
```

Listing 2: Data Structure of Function Inlining

As shown in Listing 2, class fNode has a _code_length, _children, _name, and _module. _code_length is the length of code block in SNUPL Compiler. It could be obtained by using GetInstr().size(). _children is the list of callee function. We'll explain class fCallNode after explanation of class fNode. _name is the name of the function. _module is the pointer of CModule of each function used in TAC.

Next, class fCallNode has 4 variables: _score, _loop_level, _node, and _name. _score is the score of the function call. We'll inline according to this score. It is described later. _loop_level is how many while loop is wrapped the function call. _node is the pointer of the fNode of callee function. _name is obviously the name of callee function.

4.2 Parsing Procedure of TAC

Every TAC could be parsed to above data structure. Algorithm 3 shows how to make fNode and fCallNode using TAC. Scoring the function is performed after parsing. Figure 1 is a example of original code and core data structure of parsed one.

Algorithm 3: Parsing Procedure of TAC

Data: TAC
Result: fNode
fNode = {};
forall $f \in$ *functionDeclaration* **do**
  fNode = fNode $\cup$ f;
end
forall $f \in$ *functionDeclaration* **do**
  f' = functionBody of f
  forall $fc \in$ *functionCall in* f' **do**
    fc' = new fCallNode (fc);
    fc'._loop_level = loop level of fc;
    fc'.function = functionDeclaration of fc;
    f.addChild(fc');
  end
end

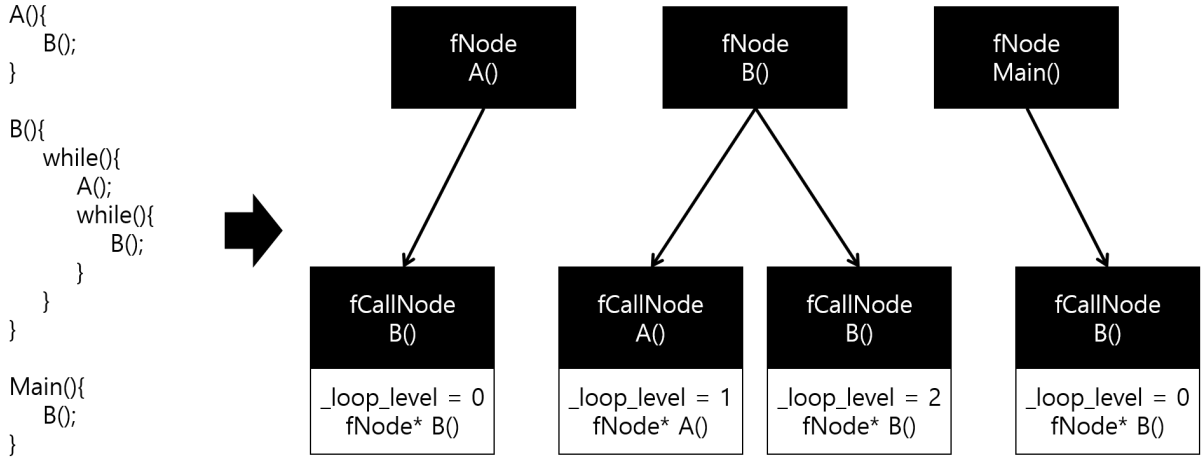

Figure 1: Constructing Data Structure of Function Inlining

4.3 Scoring The Function Call

We calculated the score each function. Obviously, inlining a frequently called and shorter function is effective. Length of code is considered in $BasicScore$ (Equation 3). $CodeLength$ is defined as length of the code block in SNUPLC. $CodeLength$ could be obtained by calling GetInstr().size() in each code block. How much it is called is considered in $LocationFactor$ and $LoopFactor$. $LocationFactor$ is that is it located in main function of subfunction. Because subfunction could be called frequently than main function, located in sub function has greater score (Equation 4). $LoopFactor$ considers how much loop wraps it. Because there is no exact way to calculate how many times the loop is repeated, we give the same score to each loop (Equation 5).

$$BasicScore = 1/CodeLength \tag{3}$$

$$LocationFactor = \begin{array}{ll} 1 & \text{(Function call is in main function)} \\ 3 & \text{(Function call is in subfunction)} \end{array} \quad (4)$$

$$LoopFactor = 5^{loop\_level} \quad (5)$$

For each function call, we could calculate $BasicScore$, $LocationFactor$, and $LoopFactor$. The final score of the function call is calculated by multiplying these three thing (Equation 6).

$$FinalScore = BasicScore \times LocationFactor \times LoopFactor \quad (6)$$

4.4 Make Function Inline

Inlining a function has 6 steps. First, we have to pick a target to inline. Then, we have to copy variables from callee function to caller function. Next, we replace param operation. Copying callee function, and replacing call operation follows next. Last, we updated parsed data.

4.4.1 Pick a Target

We inlined a function according to the score calculated in above. There could be more than one function call those have same score. In this case, we picked the earliest reached one.

4.4.2 Modify Symbol Table

To inline a callee function, we have to copy callee's local symbol table and put it on caller's local symbol table. However, if there is a same variable name in both callee's and caller's local symbol table, then it conflicts. Therefore, we have to rename copied callee's local variables. We attached prefix on copied variables. In the $(n+1)$-th inlining function, the prefix is '$in\_$'. Figure 2 shows the copied local variables. First copied variables have prefix 'i0_', and second copied variables have prefix 'i1_', and third copied variables have prefix 'i2_'. The variables which is not in local symbol table should be global variables, so we can only consider local symbol tables. With this prefix technic, we could avoid conflict of variables. When we attach prefix of each local variables, we made a hash map to find which variable had changed to what variable.

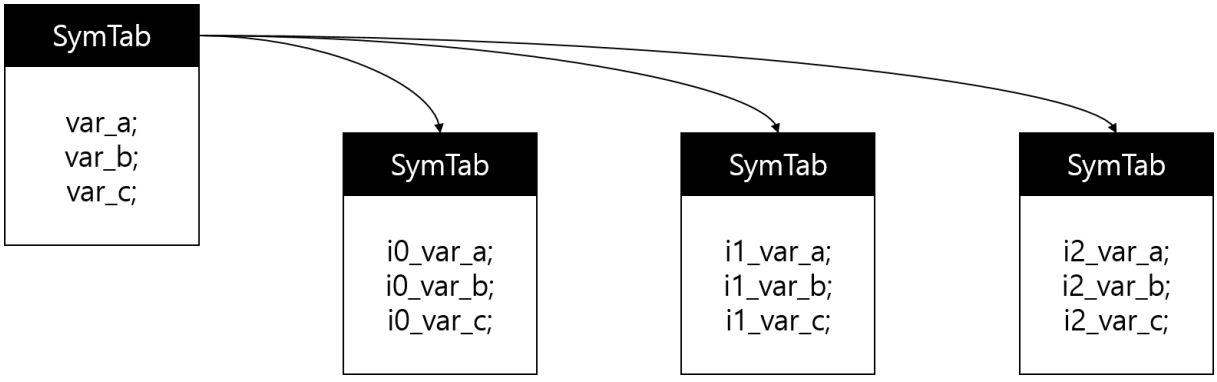

Figure 2: Copied Symtab of Function Inlining

4.4.3 Replace Param Operation

First, we have to know which parameter is connected to which callee variable. In the CSymProc of callee function, we could get i-th parameter using GetParam(i). By using hash map to find renamed variable, we could assign param operation's source to renamed variable in caller's symbol table.

4.4.4 Copy Callee Function

While we copy the callee function, we have to rename each local variable in callee's symbol table to local variable in caller's symbol table. Using hash map we made above, it is very simple problem. However, two problems are left while coping callee function. One is the return operation problem, and other is label problem.

Return operation problem could be solved by changing return operation to assign operation. Find a destination of call operation of inlining target. It there is no destination, then we can delete return operation because the program doesn't use the returned value. If there is a destination, then we assign source 1 of return operation to destination of call operation.

Label problem is that if copied label from callee's body is pasted into caller's body, there will be duplicated label after inlining same function more than one time. Therefore, making a new label using CreateLabel() function could solve the problem. Traversing from begin of the code block meets several labels. When it meets 'while_cond', make a new label '', 'while_cond', and 'while_body'. If it meets 'if_true', then make a new label '', 'if_true', and 'if_false'. Then new labels could handle every label in callee's function.

4.4.5 Replace Call Operation

Replace copied callee function with call operation in caller's function. Because return operation in callee's function is already changed to assign operation, role of call operation in callee's function is nothing left. Therefore, replacing copied callee with call operation is enough in this part. Of courese, this replacement could make number of code block go wrong, so calling CleanupControlFlow() to solve this problem is needed.

4.4.6 Update Parsed Data

After inlining a function, fCallNode children of fNode is changed. Child fCallNode which is inlined is disappeared, and new fCallNodes which are called from disappeared function are added. Deleting disppeared and adding new fCallNode is annoying, so we saved some important variables, and then called parseTAC(), and then restore important variables.

4.5 Termination Condition

Each function inlining makes code larger. Of course, if there is no more function call of specific function after inlining, then declaration of that function is useless. Removing this useless function shrinks code size. However, this optimizing technic is not our scope. Thus, more inlining makes faster running time and larger code size. If there is a loop of function call, then inlining could be performing infinitely. Therefore, we suggested the termination condition. Of course, the termination condition includes the situation that there is no remained function call to inline it.

Our termination condition is consisted of 2 condition. One is that if there is no remained function call to inline. Second is that one more inlining could make code larger than the threshold. The estimated size is calculated by adding previous size and the largest size of a function called somewhere. This function is implemented as PeekCBSize() in class Inlining. We proposed threshold as 1.5 times the original code size. However, the threshold is too small, so we increased the threshold as 2.0 times the original code size. We have classified code size as sum of code block length of every declared function. It could be obtained using GetCodeBlock()->GetInstr().size() in each function scope.

5 Evaluation

In this section, we compiled the code shown in Listing 3, and measure the runtime of compiled binary. We compared original result of SnuPLc and our 3 compilers which implemented constant propagation & dead code elimination, function inlining, and constant propagation & dead code elimination & function inlining.

```
module test17;

var input : integer;
    i : integer;
        constant : integer;
        iter : integer;
        iter2 : integer;

function sum_rec(n: integer): integer;
begin
  if (n > 0) then
    return i + sum_rec(n-1)
  else
    return 0
  end
end sum_rec;

function sum_iter(n: integer): integer;
var sum, i: integer;
begin
  sum := 0;
  i := 0;
  while (i <= n) do
    sum := sum + i;
    i := i+1
  end;
  return sum
end sum_iter;

begin
  iter := 100000000;
  iter2 := 100000000;
```

```
  while (iter > 0) do
  input := 10;
  i := 1;
  constant := 1;
  while (i>0) do
    input := constant + 1;
    if (constant > 0) then
      sum_rec(i)
        else
          sum_iter(i)
        end;
    i := i-1
  end;
iter := iter-1
  end
end test17.
```

Listing 3: Evaluation Code

The result of running time and code size is shown in Table 1. Our optimized compiler is 1.76x faster than original SNUPLc while the TAC size is 1.46x than original TAC.

	Running Time	TAC Code Size	Speed Up
Compiled with Basic SnuPLc	1,694 ms	3,112 byte	-
Constant Propagation & DCE	1,485 ms	2,825 byte	14.1%
Function Inlining	1,288 ms	5,434 byte	31.5%
Compiled with Our Compiler	960 ms	4,534 byte	76.5%

Table 1: Evaluation Result of Optimized Compiler

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[2] A. W. Appel, *Modern Compiler Implementation in C*. Cambridge University Press, 1998.