Milestone Presentation

Advanced Compiler Construction

2017-20845 Wonjae Jang
2018-25193 Dongwan Kang

October 29, 2018

1 Introduction

For four weeks after we have submitted the project proposal, we completed the implementation of Function Inlining. And rest of the compiler optimization proposed will be completed by final presentation.

In this document, we describe how we implemented Function Inlining. Those include data structure, implementing method, the result of code optimization.

2 Basic Approach

The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function through inlining. And the expanding can be implemented easily. We just scan the "opCall" operations and find the matching function and replace the statement of function call with body of the function we found. But, as mentioned in the proposal, Function Inlining usually increases code size. If the code size is too large, we have to apply Function Inlining selectively.

2.1 Function scoring

Too big code size can have a negative impact on overall performance. Thus we have to compromise, picking a number of times to expand the body of the function inline that gets the most of the performance improvement, It does not expand the code too much.

Assuming all function calls have same overhead, the importance of each "call" operation depends on the number of times called. So, we will give higher score for a function call in loop & a function call in another function scope. And we will give lower score for a function call with longer body. Each function call is scored as follows (in SnuPL).

$$\begin{aligned} Score &= 1/code\_length && \textit{by default} \\ &= 3/code\_length && \textit{in function scope} \\ &= 5/code\_length && \textit{in while scope} \end{aligned}$$

These weights are imposed intuitively, But these values will be adjusted with objective data until final presentation.

2.2 Loop & Function scope

We created new structures to store call_info of function because we need to score for consecutive function calls & repeated function call in loop properly. We save the data of call_info while scanning the TAC's code blocks. And then we can score for all function calls. And it will start to expand from the higher-scoring functions.

2.3 Inlining for Recursion

If there is a recursion, inlining expanding the calls will not terminate. So we will skip it for recursive function calls. It is easy to detect a recursion using the data structure we implemented above.

3 Implementation

3.1 Inlining

Each work of Inling consists of three processes. They are to find the target position and the code block of the function called, to replace the opCall operation of target position with source code block and to process the arguments, local variables, and returning value & destination operand. But we need to prioritize the opCalls to be replaced before we start inlining. So we traverse all of code block and create fNodes for each function and fCallNodes for each opCall. And we start inling in that order. If the total code size exceeds the limit(predefined), we do not do inling anymore.

3.2 Arguments & Variables

When replacing the function call with function body, we should transfer function arguments and local variables properly. The parameter variables and local variables in origin should be added to the caller's symbol table with a new local variable name. We should distinguish the new variables from existing ones in symbol table.

3.3 Destination Operand

If a function call has a destination operand, we should replace the opReturn instruction with a new assignment.

3.4 Data Structure

To represent functions and function calls, we made below data structure. Class fNode indicates each function including main function. Class fCallNode indicates each function call in a function. So, a fNode has a vector of fCallNodes.

```
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 1: Data structure for scoring

With this data structure, we can store the info to calculate the score and indicate the location of function call to be expanded.

3.5 Scoring

Let's consider below example.

```
void main() {
    while () {
        while () {
            f();                        // A
        }
        f();                            // B
    }
    g();                                // C
}
```

```
void f () {
    while(){
        g();                            // D
    }
}

void g(){
}
```

Listing 2: Simple Program

For this example, we have 3 fNode : main, f, g. And each fnode has 3, 1, 0 fcallNodes. fcallnode means the target position to be expanded by inling. Each fcallnode can be scored as follow.

A- f() in double loop	*5 * 5 = 25*
B- f() in single loop	*1 * 5 = 5*
C- g()	*= 1*
D- g() in single loop	*3 * 5 = 15*

In this example, we impose the highest score(25) for the function call-'A'. And function call will be expanded first.

4 Evaluation

We have made a example program that repeats multiple function calls millions of times. And the comparison result is as follow.

	running time	TAC code size
compiled with basic snuplc	5064 ms	1281 byte
compiled with ours	3931 ms	1401 byte

We just applied a function inlining technique, but the performance of running time is improved by 29%. It seems that the overhead of funcion calls may be larger than we expected.

5 Schedule

Our schedule plan is according to Table 1.

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.

Sep.	19	Project Proposal
	19 - 30	Studying SnuPL Source Code Understanding SnuPL syntax & AST & TAC translations
Oct.	1 - 24	Implementing the Basic Inline Expansion of Function Calls Additional Ideation Preparing the Source File(mod) for Testing and Making Evaluation Tool
	25 - 29	Preparing Presentation
	29	Milestone Presentation
Nov.	1 - 17	Implementing dead code elimination & constant propagation
	19 - 30	Testing, Finding Bugs, Debugging, and Fixing
Dec.	1 - 9	Evaluating the Final Implementation & Preparing Final Presentation
	10 - 12	Final Presentation
	19	Submission of Project Report and Code

Table 1: Time Table