Generation of optimal object code

1st Ivan Arkhipov
faculty of mathematics and mechanics
St Petersburg University
St Petersburg, Russia
arkhipov.iv99@mail.ru

***Abstract*—This article is related to optimizations in code generator of MIPS assembler. This work is a part of the RuC project.**

***Index Terms*—code generation, translator, optimization, MIPS**

I. Introduction

The C programming language has significant drawbacks, such as pointer arithmetic and the lack of control over array boundaries when accessing an array element. It is noteworthy that these are the shortcomings of the language itself, and not of individual compilers. These drawbacks have a negative impact on the security of the C language.

The Department of System Programming of St. Petersburg State University is developing a translator of the RuC language, which is an improved version of the C language [1]. For example, pointer arithmetic is forbidden in RuC, which makes this language more secure than C. Scientific publications have been made on this topic [2] [3]. At the moment, the RuC project has become an industrial one, which shows the relevance and practical significance of this project.

With the development of RuC, the need of implementation of optimizations arose. Without optimizations, the translator will not be able to compete with its analogs.

Generation of optimal object code is one of the main and one of the most difficult tasks in the field of system programming. This paper is devoted to optimizations in the MIPS [4] assembler, which complicates the optimization problem. MIPS is a RISC architecture, which gives a huge variability in code generation and its optimizations.

The developer of the code generator faces difficult questions. What optimizations should be implemented? What optimizations will give a significant gain in the speed of the program? Which optimizations will often speed up the program, and which will work very rarely? What architecture features should be considered when optimizing the code? All this requires an analysis of other translators, optimization approaches, and a lot of work with the machine in which codes the program is translated.

Some optimizations cannot be implemented only within the framework of the code generator. For complex optimizations, you may need to change the abstract syntax tree or add additional information to the translator tables.

After implementation, it is necessary to re-analyze other translators in order to find out where and how else you can optimize the code, how much and in what cases RuC loses to its analogs, and in what cases and by how much RuC has become better.

II. Motivation

This publication solves a practical problem. The development of the translator is impossible without the implementation of optimization. It is especially important in an industrial project. Optimizations will allow RuC to compete with its analogs in efficiency.

This article considers not only the implementation of certain optimizations, but also the analysis of approaches and analogues of RuC. This paper describes optimizations that can be implemented for C compilers as well, since the specific features of the RuC language do not affect these optimizations. Therefore, this work will be interesting to the developer of the code generator and the optimizer.

III. Problem Statement

The goal of this paper is implementation of a set of optimizations for the MIPS code generator.

To achieve this goal, the following tasks were set:

- Analysis of widespread analogs of RuC: GCC and Clang
- Creating a list of optimizations
- Implementation of optimizations in the RuC code generator
- Evaluation of results and comparison with analogues

The result of the work is the acceleration of the work of the compiled programs. Testing and measurements are presented in the chapter Evaluation.

IV. Related Work

These translators from C to MIPS codes were selected as analogs for comparison and analysis: GCC [5] and Clang [6]. This choice is due to the popularity, prevalence and a large list of optimizations of these translators.

For speed testing, a program for multiplying 200x200 matrices 1000 times was selected. The code is provided in Application 1. The size of the matrices and the number of repeats were taken so that the program execution time was not too large, so as not to wait long for the end of the test, and not too small, so that the time measurement error was slightly affected. Firstly, this test contains important language constructs, such as loops and array slices. Secondly, matrix multiplication is a widespread problem in various computing programs.

First of all, it is necessary to measure the speed of the code received after GCC and Clang translation. All tests were executed on the Baikal-T1 processor [7]. The time was measured using the time utility [8]. The measurement results are presented in Table I.

TABLE I
COMPARISON OF CLANG AND GCC

Translator	Time
Clang	0m 31.46s
GCC	0m 36.12s

Most of all, the optimization of the inner loop itself affects the running time of the program, since the commands in it are executed the most times. Analysis and comparison of internal cycles require special attention.

A. *Clang inner loop*

The code of the internal loop of the program received after the Clang translation is given in Algorithm 1. Several important optimizations are performed here: induced variables, optimizing serial branch commands, calculating the number of loop repeats before the loop body, using the delay slot, removing unnecessary inductive variables.

Algorithm 1: Clang inner loop

```
$BB0_12:
# calculating the address of a[i][k]
    addu $1, $9, $24
# slice of a[i][k]
    lw $1, 0($1)
# slice of b[k][j]
    lw $25, 0($14)
# multiplication
    mul $1, $25, $1
# addition
    addu $15, $15, $1
# increment of the induced variable
    addiu $24, $24, 4
# branch
    bne $24, $6, $BB0_12
# increment of the induced variable
    addiu $14, $14, 800
```

However, even such good code is not yet fully optimized. During the translation, an induced variable was created not for a[i][k], but for the offset relative to a[0][0] for the variables i and k. Because of this, the address a[i][k] must be calculated at the beginning of the loop, which adds one extra command in the inner loop. This method works well when there are many slices of the form array[i][k] from a large number of arrays in the inner loop. Because of this, it is not possible to create an induced variable for slice from each array, since there are not enough registers. Therefore, an induced variable is created for the offset by variables i and k, and the addresses are counted separately. But in this test, this is not necessary, moreover, a separate induced variable was created for b[k][j]. This decision of Clang is not optimal.

B. *GCC inner loop*

The code of the internal loop of the program received after the GCC translation is given in Algorithm 2. Just like in Clang, the main and most important optimizations are also implemented here. GCC is devoid of the Clang flaw mentioned above. For this program, the induced variables are implemented in the best way, for each slice its own induced variable is created. In addition, GCC has replaced the two addition and multiplication commands with a single madd command. This command multiplies the values from the two registers and adds the result of the multiplication to the special registers hi and lo [9].

Algorithm 2: Clang inner loop

```
$L8:
# increment of the induced variable
   addiu $2,$2,800
# slice of a[i][k]
   lw $5,0($3)
# slice of b[k][j]
   lw $4,-800($2)
# increment of the induced variable
   addiu $3,$3,4
# branch
   bne $6,$2,$L8
# multiplication and addition
   madd $5,$4
```

However, the execution time of the program received during the GCC translation is longer than that received during the Clang translation. As it turned out, the reason is the use of the madd command. An experiment was set up. In the object code generated by GCC, the madd command was manually replaced with two mul and addu commands, and the program execution time on the Baikal-T1 was measured. The result is shown in Table II.

TABLE II
GCC CODE WITH AND WITHOUT MADD

	Time
With	0m 36.12s
Without	0m 27.26s

To verify the results, an additional experiment was conducted. In the object code generated by Clang, two mul and addu commands were manually replaced with one madd command, and the execution time of the program code was measured. The result is shown in Table III.

TABLE III
CLANG CODE WITH AND WITHOUT MADD

	Time
With	0m 37.18s
Without	0m 31.46s

Obviously, the madd command should not be used for optimization.

There are several approaches to implementation of optimizations. It is necessary to consider them and choose the appropriate one.

C. Implementation based on SSA-form

Many modern translators use an intermediate representation in the form of SSA to implement optimizations [5] [6]. SSA-based optimizations are also demonstrated in theoretical books [10] [11]. A lot of scientific papers are devoted to optimizations in SSA form [12] [13].

The SSA form can be either high-level or low-level, which gives this representation flexibility. For translation from high-level languages, several SSA representations can be created, which allows to implement optimizations at different levels of abstraction. The low-level representation is well suited for implementing machine-dependent optimizations.

However, the disadvantage of this approach is its complexity. At first the SSA form of the intermediate representation should be developed, then the code generator into the codes of this representation and the code generator from the codes from this representation into the codes of the real machine should be implemented, and the optimizer may be implemented.

D. Implementation based on AST-form

The second form of intermediate representation is an abstract syntax tree. It is a high-level representation and displays the hierarchical structure of programs in the translated programming language. This form is poorly suited for low-level optimizations, and, as a result, it is not enough for efficient translation of high-level programming languages.

However, the advantage of this approach is its simplicity. There is no need to develop an SSA form, write a code generator to its codes and from its codes to the codes of the target machine. Due to the simplicity, the compilation speed increases, this advantage is used, for example, in TCC [14].

V. General Design

Based on the results of the review of analogs, it was decided to implement the following optimizations:

- Optimizing serial branch commands
- Calculating the number of loop repeats before the loop body
- Using the delay slot

In RuC, there is currently no need for an intermediate representation in the SSA form. RuC is a low-level programming language, so there is no need to create complex forms of intermediate representation for analyzing and optimizing program constructs.

The selected optimizations can be implemented directly in the code generator without changing the abstract syntax tree. In the process of parsing the tree, depending on the optimization, flags is set to generate certain commands specific to each optimization. For example, the following sections will describe how to optimize the calculation of the number of loop repeats. It should be generated a comparison command for two registers instead of a comparison command with zero, as it was before optimization. To do this, during the processing of the TFor node, a flag is set to generate such commands.

VI. Implementation

A. Optimizing serial branch commands

During the code generation of loops, it is possible to form unnecessary branch commands that create a sequence of jumps. Let's consider an example of a loop generated by the RuC translator. The loop code is shown in Algorithm 3.

Algorithm 3: RuC inner loop before optimization

```
BEGLOOP23:
# calculating a condition
   addi $t1, $s2, -200
# branch
   bgez $t1, ELSE22
# loop body
   ...
CONT23:
# increment of the inductive variable
   addi $s2, $s2, 1
# jump
   j BEGLOOP23
ELSE22:
# code after loop
   ...
```

At first, the loop condition is calculated, then it is checked, and at the end, the jump to the calculation and checking of the condition is executed. This loop can be optimized by setting a condition and a branch command before the loop and at the end of the loop. The optimized code is presented in Algorithm 4.

This optimization is implemented in the code generator in the processing module of the TFor node and conditional expression nodes. In the TFor node processing module, the loop structure is organized in the appropriate way. In the module for processing conditional expression nodes, the appropriate branch commands are generated.

Algorithm 4: RuC inner loop after optimization

```
# calculating a condition
   addi $t1, $s2, -200
# branch
   bgez $t1, ELSE26
BEGLOOP27:
# loop body
   ...
CONT27:
# increment of the inductive variable
   addi $s2, $s2, 1
# calculating a condition
   addi $t1, $s2, -200
# branch
   bltz $t1, BEGLOOP27
ELSE26:
# code after loop
   ...
```

To check the correctness, tests were run for various conditions for exiting the for loop [15]. This paper does not consider a formal proof of the correctness of optimizations. It is enough to pass prepared tests.

B. Calculating the number of loop repeats before the loop body

When generating code for loops, it is better to generate the code of the exit condition before the loop body. The condition will be evaluated before the loop and will not be evaluated in each iteration of the loop. It is necessary to remember the condition in the register, and at the end of the cycle compare the value of the inductive variable with this register. Code without this optimization is shown in Algorithm 4 and optimized code is shown in Algorithm 5.

This optimization requires allocating a register to store the condition. Due to the limited number of registers, optimization is implemented only for the most internal loops. This optimization is implemented in the code generator in the processing module of the TFor node and conditional expression nodes. In the TFor node processing module, the loop structure is organized in the appropriate way. In the module for processing conditional expression nodes, the appropriate branch commands are generated.

There may be situations when the loop exit condition is too complex, so it is not possible to apply this optimization. The possibility of applicability will be considered at earlier stages of translation and is beyond the scope of this work.

To check the correctness, tests were run for various conditions for exiting the for loop [16].

Algorithm 5: RuC inner loop after optimization

```
# calculating and saving a condition
   addi $s5, $0, 200
# branch
   bge $s2, $s5, ELSE26
BEGLOOP27:
# loop body
   ...
CONT27:
# increment of the inductive variable
   addi $s2, $s2, 1
# branch
   bne $s2, $s5, BEGLOOP27
ELSE26:
# code after loop
   ...
```

C. Using the delay slot

A delay slot is an instruction slot being executed without the effects of a preceding instruction. There are branch delay slot in MIPS architecture [9]. This is a feature of pipelined computing.

The MIPS directive ".set reorder" [17] and the GCC assembler option "-mcompact-branches=optimal" [18] allow to fill the delay slot with a previous command when it is possible. If this is not possible, the nop command is inserted. A command can be inserted into the delay slot only if it does not work with the registers that are used in the branch command.

Algorithm 6: RuC inner loop after optimization

```
# calculating and saving a condition
   addi $s5, $0, 200
# branch
   bge $s2, $s5, ELSE26
# decrement of the inductive variable
   addi $s2, $s2, -1
# decrement of the condition
   addi $s5, $s5, -1
BEGLOOP27:
# increment of the inductive variable
   addi $s2, $s2, 1
# loop body
   ...
CONT27:
# branch
   bne $s2, $s5, BEGLOOP27
# increment of the condition
   addi $s5, $s5, 1
ELSE26:
# code after loop
   ...
```

The essence of optimization is to allow GCC to fill the delay slot. It is necessary to move the increment of the inductive variable to the beginning of the loop and reorganize loop code. Code without this optimization is shown in Algorithm 5 and optimized code is shown in Algorithm 6.

This optimization is implemented in the code generator in the processing module of the TFor node. In the TFor node processing module, the loop structure is organized in the appropriate way.

To check the correctness, tests were run for various conditions for exiting the for loop [19].

VII. Evaluation

To demonstrate the effectiveness of optimizations, a program for multiplying 200x200 matrices 1000 times was selected. The code is provided in Application 1. The size of the matrices and the number of repeats were taken so that the program execution time was not too large, so as not to wait long for the end of the test, and not too small, so that the time measurement error was slightly affected. Firstly, this test contains important language constructs, such as loops and array slices. Secondly, matrix multiplication is a widespread problem in various computing programs.

All tests were executed on the Baikal-T1 processor [7]. The time was measured using the time utility [8]. The measurement results are presented in Table IV.

TABLE IV
RuC optimizations

	Time
Without optimizations	1m 37.91s
Optmization of serial branch commands	1m 34.66s
Previous + Optmization of condition calculation	1m 32.13s
Previous + Using delay slot	1m 29.74s

The operating time of the matrix multiplication program has become significantly less. However, the result is still far from the RuC analogues (Table V), since there aren't many optimizations in the RuC compiler.

TABLE V
Comparison of Clang, GCC and RuC

Translator	**Time**
Clang	0m 31.46s
GCC	0m 36.12s
RuC	1m 29.74s

For further implementation, there are optimizations related to induced variables and reduction of the inductive variable. There is an hypothesis that the induced variables will give a big profit in time, since many commands are spent on slices from the array, but this is already a theme for future work.

VIII. Conclusion

This paper solves the problem of optimizations in the MIPS architecture in the code generator. The GCC and Clang compiler were analyzed. Based on the analysis, optimizations for implementation were selected: optimizing serial branch commands, calculating the number of loop repeats before the loop body and using the delay slot. The efficiency of optimizations was shown by the example of matrix multiplication. Thus, the code generation for loops was optimized in the RuC translator.

This work has many opportunities for further research. Optimization is a very broad topic for research. In this paper, the result of analogs for the matrix multiplication test has not yet been achieved. The research can be continued with the implementation of induced variables and reduction of inductive variable. There are many other language constructions, the optimization of which is not considered in this work. These are, for example, function declarations and calls, or arithmetic operations. As you can see, there are still many sources for research.

References

[1] RuC project github – URL: https://github.com/andrey-terekhov/RuC (accessed: 04.04.2021)

[2] Terekhov A. N., Terekhov M. A. "RuC project for education and reliable software systems development" ISSN 0321-2653 Izvestiya Vuzov. Severo-Kavrfzskiy Region. Technical Science, 2017

[3] Arkhipov I.S. Code generation for floating-point arithmetic in architecture MIPS. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). 2020;32(3):49-56

[4] System V Application Binary Interface MIPS RISC Processor, 3rd Edition

[5] GCC official site – URL: https://gcc.gnu.org/ (accessed: 04.04.2021)

[6] LLVM official site – URL: https://llvm.org/ (accessed: 04.04.2021)

[7] Baikal-T1 specifications – URL: http://www.baikalelectronics.ru/products/35/ (accessed: 04.04.2021)

[8] time(1) Linux manual page – URL: https://man7.org/linux/man-pages/man1/time.1.html (accessed: 04.04.2021)

[9] MIPS Architecture for Programmers Volume II-A: The MIPS32 Instruction Set Manual

[10] Aho, Alfred Vaino; Lam, Monica Sin-Ling; Sethi, Ravi; Ullman, Jeffrey David (2006). Compilers: Principles, Techniques, and Tools (2 ed.)

[11] S. Muchnick. Advanced Compiler Design and Implementation, 1997

[12] Kathleen Knobe, Vivek Sarkar. Array SSA form and its use in parallelization. POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 1998

[13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, Steve Arthur Zdancewic. Formal verification of SSA-based optimizations for LLVM. PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2013

[14] Tiny C Compiler Reference Documentation – URL: https://bellard.org/tcc/tcc-doc.html (accessed: 04.04.2021)

[15] Tests for optimization of serial branch commands – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/cycle_jump_reduce (accessed: 04.04.2021)

[16] Tests for optimization of condition calculation – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/cycle_condition_calculation (accessed: 04.04.2021)

[17] MIPS Assembly Language Programmer's Guide

[18] GCC MIPS options – URL: https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html (accessed: 04.04.2021)

[19] Tests for optimization of delay slot – URL: https://github.com/IvanArkhipov1999/RuC/tree/mips/tests/mips/optimizations/delay_slot (accessed: 04.04.2021)

APPLICATION 1

```
void main()
{
        int a[200][200], b[200][200], c[200][200];
        register int i, j, k, v;
        for (i = 0; i < 200; ++i)
        {
                for (j = 0; j < 200; ++j)
                {
                        a[i][j] = i * j;
                }
        }

        for (i = 0; i < 200; ++i)
        {
                for (j = 0; j < 200; ++j)
                {
                        b[i][j] = i + j;
                }
        }

        for (v = 0; v < 1000; ++v)
        {
                for(i = 0; i < 200; ++i)
                {
                    for(j = 0; j < 200; ++j)
                    {
                        register int cij = 0;
                        for(k = 0; k < 200; ++k)
                            cij += a[i][k] * b[k][j];
                        c[i][j] = cij;

                    }
                }
        }
        printf("%i\n", c[0][0]);
}
```