Optimizing Method of C Compiler for TRON Architecture

Seiji Hayashida
Kiichiro Tamaru

Advanced Microprocessor Technology Department
Semiconductor Device Engineering Laboratory
TOSIBA Corporation, 580-1, Horikawa-cho, Saiwai-ku, Kawasaki, 210 JAPAN

Abstract

This paper describes the optimizing methods used in the ANSI-C compiler for the chip based on the TRON architecture. This C compiler is designed for the TLCS-90000/TX series microprocessors. For the C compiler, unique optimizing methods for the TRON architecture are used in the routines for optimization of intermediate language and code generation, in addition to the traditional global optimizing methods, such as copy propagation, loop optimization and register calling convention. Thus the compiling performance was improved. The unique features of the TRON architecture are: chained addressing mode, and the ACB and SSTR instructions. Finally, the performance of the optimizing compiler is evaluated in the terms of the execution time and object code size.

1 Introduction

The C compiler mentioned in this paper is developed by Toshiba to be used for the TRON specification 32-bit micro-processor TLCS-90000/TX series [1] [2]. This C compiler has the following features:
2. Optimization for reducing execution time and code size
3. Extended functions for embedded control systems

This C compiler employs the optimizing methods for efficiently using the unique features of the TRON specification chips, as well as traditional global optimizing. This paper describes these optimizing methods.

Subsequent sections in this paper are pertinent to the following topics: Section 2 explains about the configuration of the C compiler and the processing flow of the code generator which executes optimization. Section 3 describes three conventional optimizing methods: global optimizing, register calling convention, and inline expansion. Section 4 describes the chained mode, and the ACB and SSTR instructions specific to the TRON specification chips and gives their optimizing methods. Section 5 gives the results of comparing the program execution times, object sizes, and compile times for the optimizing methods. Section 6 gives a summary on this paper.

2 C Compiler Configuration

2.1 Entire C Compiler Configuration

The C compiler consists of three programs: the preprocessor, intermediate code generator, and code generator. These programs are linked to each other by files. The preprocessor executes preprocessing on the C language. The intermediate code generator carries out lexical analysis, syntax analysis, and intermediate code generation. The code generator performs code optimization and code generation. By these programs, source text files in the C language are compiled into those in the assembly language.

2.2 Code Generator Processing Flow

Figure 1 shows the flow of the code optimization and code generation by the code generator:

In Intermediate code input, intermediate code and symbol information are input from the intermediate code file and expanded in memory. In Intermediate code optimization, the intermediate code
is optimized by global optimizing, etc. In Storage allocation, register allocations are executed considering the variable's lifetime. In this stage, variables, which are failed or prohibited to allocate to registers, are allocated to the memory on the stack. In Assembly code generation and optimization, assembly code is generated from the intermediate code. The optimized instructions are selected during generation by peephole optimization in the assembly code level. In Assembly code output, the assembly code generated in memory is output to the file.

The code generator carries out the operations above in the units of functions to generate assembly code from the intermediate code.

3 Traditional Optimizing Methods

3.1 Global Optimizing

The traditional global optimizing methods supported are as follows: [4]

- Copy propagation and constant folding
- Common subexpression elimination
- Loop optimizing

Because the optimizing methods above are independent of the processor, they are available in the stage of intermediate code.

3.2 Register Calling Convention

Most C compilers pass function arguments through the stack, since the C language supports variable arguments functions. However, there are only a few cases where variable arguments functions are required. This C compiler allows use of the register calling convention which passes arguments as registers for functions not using variable arguments. The register calling convention requires fewer accesses to the stack than the stack calling convention which passes arguments through the stack, therefore, provides higher execution speed. The register calling convention also requires less stack size. The following sample program shows how to use the register calling convention:

C program:
```c
int a;
void test() {
    add(a,1);
}
int add(int i, int j) {
    return i+j;
}
```

Assembly program after compiled in the stack calling convention:
```assembly
.test:    mov #1, @sp
         push @a
         bsr .add
         add #8, sp
         rts
.add:    mov 0(sp), r0
         add 0(sp), r0
         rts
```

Assembly program after compiled in the register calling convention:
```assembly
.test:    mov @a, r1
         mov #1, r2
         bsr $add
         rts
$add:    mova @0(r1,r2), r0
         rts
```

In the example above, four memory accesses are eliminated in total for both argument read and write, and the stack size is also reduced by eight bytes.

3.3 Inline Expansion

The inline expansion is a method of generating instructions which form a body of the functions to be invoked instead of function call instructions. Use of the inline expansion can reduce the execution time because it does not require branching and argument
passing necessary for function calls. But, if the same function call occurs more than once in the same source, the inline expansion is executed for each occurrence, resulting in the increased object size. Since this C compiler has been designed for embedded control systems, the code size of the generated object must be minimized. This C compiler supports the inline expansion feature for functions, but it is left to the user when to execute inline expansion on which function. The user can specify the function to be inline-expanded and locally enable or disable the inline expansion.

4 TRON Chip Specific Features and Optimizing Methods

The TRON specification chips are CISC-type processors, and support high-performance instructions applicable to compilers, and addressing modes. This section describes the chained mode, the SSTR instruction, and the ACB instruction used by the C compiler for optimizing [5] [6] [7].

4.1 Chained Mode

The chained mode allows to handle addressing operations for adding, scaling, and an indirect reference as one operand by combining them in multiple stages. In the TX series, this mode is supported to a maximum of four stages. In this addressing mode, the initial value is set in a register or set as 0, and a numeric value and register*4 are added to calculate the address for memory read. The memory can be further accessed using the above memory read result as the intermediate result and the sum of a numeric value and contents in a register as the address. This process can be specified in an operand.

The following sample program shows how to use the chained mode:

C program:
```c
int i, j, k;
extern char array[100];
return array[i+j*2+k*4];
```

Assembly program after compiled:
```assembly
mov @(array,r4*4),r5
```

The accessing part of the array forms an operand for chained mode, which specifies accessing to the memory at address

(r4+r5*2+array+r6*4).

Without the chained mode, ten instructions shown below would be necessary:

```assembly
mov #array,r0
mov r6,r2
shl #1,r2
mov r5,r1
add r2,r1
mov r4,r2
shl #2,r2
add r2,r1
add r1,r0
mov @r0,b,r0
```

Thus, one chained mode instruction can substitute ten instructions. The object size is reduced from 24 bytes to 12 bytes, the execution time is reduced from 1.25 \(\mu\)S to 0.4 \(\mu\)S (@TX1 20MHz), and the register for address calculation becomes unnecessary.

4.2 Optimizing Method for Chained Mode

This addressing mode is generated by searching for use-definition chain of temporary variable in the process of Assembly code generation. The following example shows the intermediate code and the chained mode generated.

C program:
```
j = array[i];
```

Intermediate code:
```
T0 = &array
T2 = i * #4
T1 = T0 + T2
j = *T1
```

Register allocation:
```
T0:r0, T1:r0, T2:r1, i:r4, j:r5
```

Assembly program after compiled:
```
mov @(array,r4*4),r5
```

In the intermediate code above, three statements are necessary to calculate the array[i] address. "T0 = &array" calculates the address allocated to the variable array, "T2 = i * #4" multiplies the variable i by 4 because an element of the array uses four bytes, and "T1 = T0 + T2" calculates the sum of variables T0 and T2 to get the address indicated by array[i]. "j = *T1" reads data from memory at the address indicated by the variable T1 calculated by the three statements above, then assigns the read data to the
variable j. Now a problem arises that more than one intermediate code statement is required to generate one chained mode. In the processing flow, Assembly code generation comes after Storage allocation, which means that the register allocated to the temporary variable of the intermediate code in the chained mode is not used for the assembly code, wasting the register. In the example above, registers r0 and r1 allocated to variables T0, T1, and T2 are not used in the compiled result.

To avoid wasting registers and to make the register allocation efficient, a single-stage chained mode which occurs frequently should be generated in the stage of the intermediate code. The intermediate code with the chained mode generated is as follows:

Optimized intermediate code:
\[ j \times (*(&array, i*4)) \]

This intermediate code specifies accessing array[i] by its operand *(&array, i*4).

However, chained mode using multiple stages is generated in Assembly code generation because it is too much complicated to be processed in the intermediate code level and it would disturb the other code optimization. In this case, some registers are left unused.

### 4.3 ACB Instruction

The ACB instruction performs addition, subtraction, and conditional branch operations by one instruction. (There is also the SCB instruction which performs a subtract operation instead of an add operation.) The ACB instruction has the following specifications:

**Mnemonic and operand:**
\[ ACB \ step, xreg, limit, offset \]

**Instruction operation:**
\[ xreg += step \]
\[ if(xreg < limit) \]
\[ goto PC+offset \]

A sample program using the ACB instruction is given below.

**C program:**
```c
int i;
for(i=0;i<10;++i)
;
```

**Assembly program after compiled:**
```assembly
mov #0.r4
L0:
acb #1.r4,#10,L0
```

In the example above, when the ACB instruction is not used, the following three instructions would be required:
```assembly
add #1.r4
cmp #10.r4
blt L0
```

Thus, the ACB instruction requires only one instruction instead of three. It reduces the object size from 8 bytes to 4 bytes, and the execution time from 5.3 $\mu$S to 4.7 $\mu$S (@TX1 20MHz).

### 4.4 Optimizing Method for the ACB Instruction

The ACB instruction is generated by searching for the instruction string for addition, subtraction, and conditional branch during the peephole optimization processing in the process of Assembly code generation and optimization. Though the ACB instruction seems convenient, it puts some operational restrictions. Two functional restrictions are:

1. Only the less than condition is available for compare.
2. The vreg operand used for addition and comparison is available only for registers.

Taking the execution time and object size into consideration, only the Q and R formats are available as instruction formats. Because of restrictions on the operands used for these two instruction formats, the number of integers or registers is limited to 1 for step (counter value for add operation) and to 1-64 for limit (compare object). The following example shows an unavailable program:

**With the ACB instruction**
```assembly
mov #0.r4
L0:
acb #2.r4,#20,L0
```

**Without the ACB instruction**
```assembly
mov #0.r4
L0:
add #2.r4
cmp #20.r4
blt L0
```
In this program, the ACB instruction has the disadvantages that it increases the object size by two bytes and the execution time from 5.3 μS to 5.8 μS (at TX1 20MHz). Thus, when generating the ACB instruction, it must satisfy some conditions for improving the performance.

### 4.5 SSTR Instruction

The SSTR instruction fills the memory area with predetermined values. It sets the start address, length, and the parameters of the initial value in the registers before execution. This single instruction allows the contiguous memory area to be speedily initialized. The SSTR instruction performs the following operation:

**SSTR instruction operation:**

```
do {
  --r2;
  *ri = r3;
  ri += size;
} while (r2!=0);
```

This instruction writes the contents of register `r3` into the memory indicated by register `ri` for the number of times specified by register `r2`. When initializing an array with a for statement, the compiler can generate the SSTR instruction. The SSTR instruction can also be generated when initializing a multi-dimensional array by a nested loop as shown in the following example:

**C program:**

```c
for(i=0;i<i0;++i)
  array[i] = i;
```

**Assembly program after compiled:**

```
mov #array, RI
mov #1, r3
mov #100, r2
sstr.w
```

Without the SSTR instruction, the compiled result is as follows:

```
mov #0, r4
mov #a, r1
L4:
mov #0, r5
mov r1, r0
L8:
mov #1, r0
add #4, r0
acb #1, r5,#10,L8
add #40, r1
acb #1, r4,#10,L4
```

In this program, the SSTR instruction shortens the memory initializing time to approximately \( \frac{1}{5} \) (from 74.8 μS to 15.6 μS @TX1 20MHz) and reduces the object size from 26 bytes to 14 bytes.

### 4.6 Optimizing Methods for SSTR Instruction

The SSTR instruction uses fixed registers `r1` to `r3` as parameters. Therefore, registers `r1` to `r3` cannot be allocated to a variables which steps over the SSTR instruction. Registers allocated to variables are decided considering whether these registers are allocatable or unallocatable. In addition, the intermediate code corresponding to the SSTR instruction must be generated in the **Intermediate code optimization** because the register allocation process must be able to recognize which intermediate code is the SSTR instruction. In this C compiler, the intermediate code corresponding to the SSTR instruction is generated by searching for the loop intermediate code equivalent to the SSTR instruction during loop optimization in the **Intermediate code optimization**. The following example shows a case the SSTR instruction is generated:

**C program:**

```c
for(i=0;i<i0;++i)
  array[i] = i;
```

**Intermediate code:**

```
T0 = array
i = #0
L4:
  *T0 = #1
  T0 += #4
  i += #1
  if i < #10 goto L4
```

**Intermediate code after generating the SSTR instruction:**

```
T0 = &array
SSTR T0,#1,#10
```

**Assembly program after compiled:**

```
mov &array, R1
mov #1, r3
mov #10, r2
sstr.w
```
In the example above, the six instructions in the loop intermediate code ("#1 = #0", "#L4", "*#TO = #1", "#TO += #4", "#1 += #1", and "if #1 < #IO goto #L4") specify writing constant "#1" 10 times from the start address #TO. The SSTR instruction substitutes these six instructions. For a nested loop, the processing scope of the SSTR instruction can be expanded by substituting the SSTR instruction from the inner-most loop.

5 Performance Evaluation

This section gives the comparison results of the execution time and object size produced by Dhrystone benchmark program [8]. The execution environment to get the execution time is: processor: TX1, clock frequency: 20MHz, memory wait: 0 wait. The compile time means the time required for the code generator processing in the SPARC station 11. The compare results shown in Table 1.

<table>
<thead>
<tr>
<th>Compile mode</th>
<th>execution time (m Seconds)</th>
<th>Object size (Bytes)</th>
<th>Compile time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack calling</td>
<td>1,006</td>
<td>1,310</td>
<td>0.7</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack calling</td>
<td>894</td>
<td>1,130</td>
<td>1.4</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register calling</td>
<td>835</td>
<td>1,094</td>
<td>1.4</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register calling</td>
<td>782</td>
<td>1,062</td>
<td>1.4</td>
</tr>
<tr>
<td>Specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Evaluation Stanford

<table>
<thead>
<tr>
<th>Compile mode</th>
<th>execution time (m Seconds)</th>
<th>Object size (Bytes)</th>
<th>Compile time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack calling</td>
<td>2,279</td>
<td>2,124</td>
<td>1.2</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack calling</td>
<td>1,255</td>
<td>1,474</td>
<td>2.5</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register calling</td>
<td>1,236</td>
<td>1,428</td>
<td>2.5</td>
</tr>
<tr>
<td>No specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global optimizing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register calling</td>
<td>1,196</td>
<td>1,132</td>
<td>2.5</td>
</tr>
<tr>
<td>Specific feature</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In optimizing specific features of the this program, 43 chained modes, 28 ACB instructions, and 2 SSTR instructions were generated. This program could reduce the execution time by approximately 46% and the object size by approximately 33% by global optimizing and register calling convention. In addition, optimizing on specific features reduced the execution time by approximately 3% and the object size by approximately 21%. The execution time can also be reduced on a RISC-type processor by global optimizing and register calling convention, but reduction of the execution time by optimizing the specific features is only enabled on a CISC-type processor. In this evaluation, though the execution time could not be significantly reduced, the object size could be reduced approximately 20% in Puzzle. The chained mode specific to the TRON specification chips made an important contribution to the reduction of the object size. Global optimization processing requires a long time, resulting in an increased compile time, but the register calling convention and specific feature optimizing do not cause the processing time to be increased.

6 Summary

This paper described the configuration of the C compiler developed first, then the global optimizing, register calling convention, and inline expansion as conventional optimizing methods. It then described
the chained mode, the ACB and SSTR instructions, and optimizing methods for these instructions. In these methods, register allocation is made efficiently by using chained mode at intermediate language level. The ACB instruction was generated considering the conditions for efficiently using this instruction. The SSTR instruction could reduced the execution time and object size by adding operations which substitute this instruction for the instructions in a loop. Finally, we made performance evaluation using optimizing method by comparing the program execution time, object size, and compile time. The execution time and object size could be reduced most efficiently by global optimizing, but it increased the compile time. The register calling convention and specific function optimizing could reduce the execution time and object size without increasing the compile time. Especially for specific functions of the TRON specification chips, the chained mode contributed most to the improvement of performance.

When the optimizing method is used which supports the processor specific functions from the stage of the intermediate code, the execution time and object size can be reduced without significantly increasing the processing time.

Acknowledgement

The authors would like to thank Dr. Ken Sakamura of University of Tokyo for his many helpful suggestions and the members of the TRON Association for their cooperation in standarding the TRON specification.

References