2009-01-0149

Techniques for Generating and Measuring Production Code Constructs from Controller Models

Bill Chou, Saurabh Mahapatra
The MathWorks

ABSTRACT

A key step in Model-Based Design is the deployment of an algorithm as machine code onto a target processor in the production vehicle. Modern software tools automatically generate the algorithmic source code from models. Given the many combinatorial possibilities for realizing a given algorithm within the modeling environment, the generated C source code will be a function of a realization. This dependency is an important consideration because the quality and clarity of the source code impacts the amount of verification and analysis that must be done for production software development. Other factors involved in generating the machine code from the source code, such as compiler optimization and microprocessor architecture, also contribute to this optimization. Organizations that proactively data mine and gather these optimizations into a set of best practices stand to benefit from reduced development times and lower costs. This paper introduces techniques that can be used to generate and measure code constructs used to create a set of best practices for the Simulink modeling environment. The quality of the object code is measured by examining the algorithm compiled within an Integrated Development Environment.

MODEL-BASED DESIGN

Model-Based Design for embedded control systems development involves a process centered on a model—from requirements capture to implementation and test. This model forms the “executable specification” that is used to communicate the desired system performance. The control design is elaborated and continuously tested against requirements through simulation. Code is generated from models and rapid-prototyping is carried out to assess the performance of the algorithm in a real-time environment. Software-in-the-loop (SIL), processor-in-the-loop (PIL), and hardware-in-the-loop (HIL) testing and verification of the algorithmic code may be done before deployment on the production vehicle.

The use of automatic code generation maintains the link between the model and the generated C source code[1]. To change the algorithm later in the design process, it is easier to update the model and regenerate the C source code. This method allows the engineer to focus more on integrating algorithmic code and setting up the infrastructure for embedded system deployment[2].

Figure 1 shows the code generation workflow in Model-Based Design. A and B denote opportunities for optimizing code.

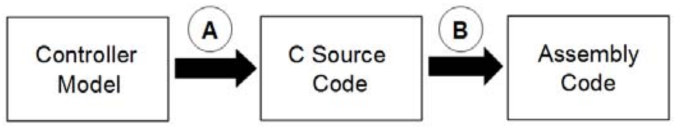


Figure 1. Code generation workflow in Model-Based Design.

For each opportunity, several techniques are available:

A. Generating C source code from software models:
   - Using modeling design patterns in the controller model
   - Using target-optimized code

B. Compiling C source code into object code:

SAE Customer Service:	Tel:	877-606-7323 (inside USA and Canada)
	Tel:	724-776-4970 (outside USA)
	Fax:	724-776-0790
	Email:	CustomerService@sae.org
SAE Web Address:	http://www.sae.org	

Printed in USA

- Choosing a microprocessor architecture
- Choosing a compiler
- Using compiler-specific optimizations

Two important metrics to measure the quality of control algorithms running on microprocessors are object code size and execution time.

Object code size is used to measure the quality of the control algorithms, although the proposed techniques can save time as well[3]. This is due to difficulties in profiling object code. One can look at either the execution time or speed. If time is being measured over several trials, the variability requires looking at the minimum, maximum, or average execution times. If efficiency is being measured by throughput, it is measured differently from execution time. Hence, we use object code size to measure the quality of code.

The following examples illustrate the application of these techniques to optimize object code. Real-Time Workshop Embedded Coder is used to automatically generate C source code from Simulink models. The code is compiled and loaded onto processors supported by Green Hills MULTI. Standard code generation optimization settings, such as expression folding and block reduction, and compiler flags, such as `–a` and `-Osize`, were used unless stated otherwise.

GENERATING C SOURCE CODE FROM SOFTWARE MODELS

Two techniques are available for optimizing the C source code generated from the software model: modeling patterns and target-optimized code.

USING MODELING DESIGN PATTERNS IN THE CONTROLLER MODEL - A modeling design pattern is much like a software design pattern used in object-oriented literature[4]. It is a template containing modeling elements that can be reused in commonly recurring design problems. Figure 2 shows an example of a Stateflow modeling design pattern for the familiar `do-while` logic. This pattern can be used to generate the common `do-while` loop construct in the C code.

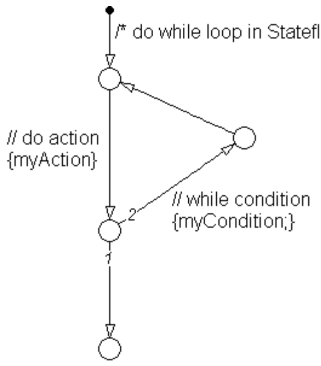


Figure 2. Stateflow `do-while` loop design pattern.

We are interested in modeling design patterns that optimize C source code measured by lines of code (LOC). At a high level, it may lead to more optimized object code.

Figure 3 shows the matrix multiplication of two 10x10 matrices `u_1` and `u_2` in Stateflow. The outer two loops use counters `i` and `j` to loop through rows of `u_1` and columns of `u_2`. The inner-most loop computes each element of the output matrix `y_1` as the dot product of the row from `u_1` and the column from `u_2`. The model uses nested loops very similar to the Stateflow `do-while` loop design pattern shown in Figure 2. The difference lies in the duplicate initializations of `y_1[i][j]` in the outer `i` and `j` loops.

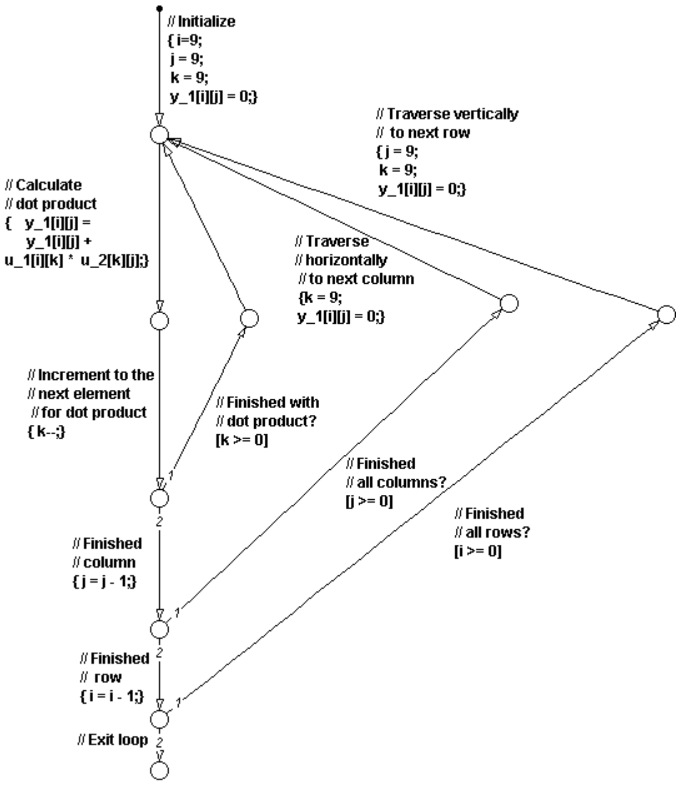
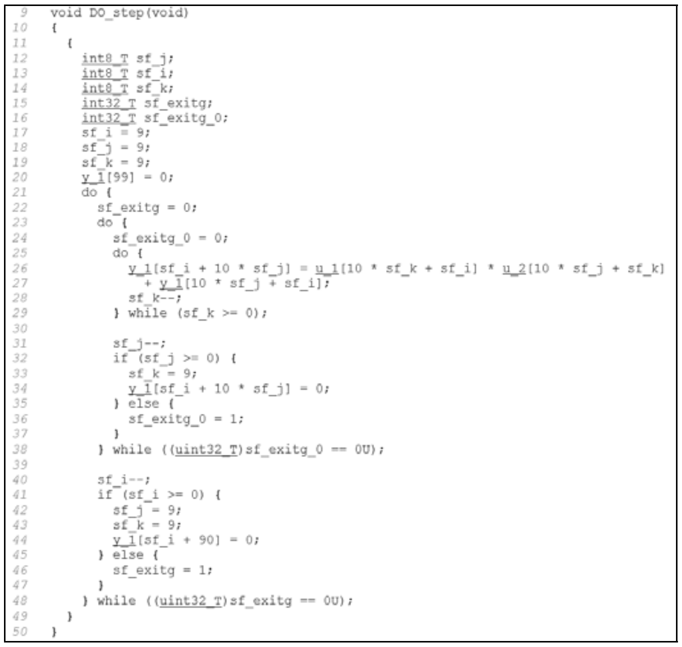


Figure 3. Modeling multiplication of two 10x10 matrices without use of a modeling design pattern.

Figure 4 shows 42 LOC generated from this model. Note the checks for `i` and `j` with redundant initializations of `y_1[i][j]` on lines 32–37 and 41–47. These multiple initializations can be reduced to just one initialization before the `do-while` loop in lines 25–29.



```
void DO_step(void)
{
  {
    int8_T sf_j;
    int8_T sf_i;
    int8_T sf_k;
    int32_T sf_exitg;
    int32_T sf_exitg_0;
    sf_i = 9;
    sf_j = 9;
    sf_k = 9;
    y_1[99] = 0;
    do {
      sf_exitg = 0;
      do {
        sf_exitg_0 = 0;
        do {
          y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
            + y_1[10 * sf_j + sf_i];
          sf_k--;
        } while (sf_k >= 0);

        sf_j--;
        if (sf_j >= 0) {
          sf_k = 9;
          y_1[sf_i + 10 * sf_j] = 0;
        } else {
          sf_exitg_0 = 1;
        }
      } while ((uint32_T)sf_exitg_0 == 0U);

      sf_i--;
      if (sf_i >= 0) {
        sf_j = 9;
        sf_k = 9;
        y_1[sf_i + 90] = 0;
      } else {
        sf_exitg = 1;
      }
    } while ((uint32_T)sf_exitg == 0U);
  }
}
```

Figure 4. Generated C source code without the use of modeling design patterns.

Figure 5 shows an implementation of the same algorithm that makes proper use of a nested Stateflow `do-while` loop design pattern. The generated C source code (see Figure 6) has only 25 LOC compared with the 42 LOC shown in Figure 4. The redundant initializations of `y_1[i][j]` and checks for `i` and `j` have been eliminated, resulting in more efficient C source code.

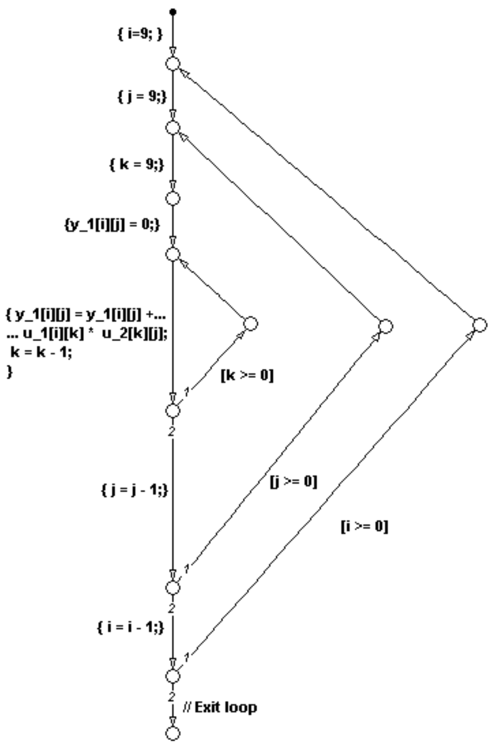
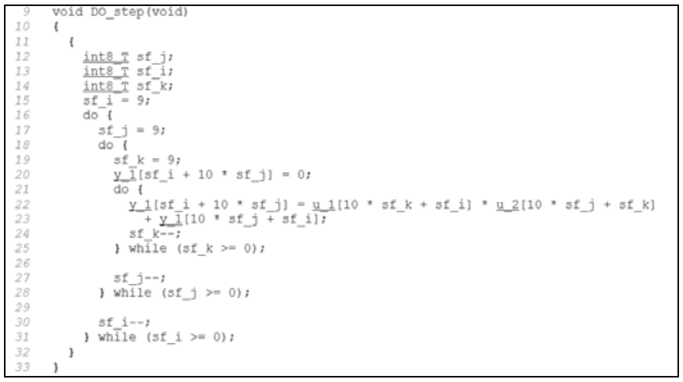


Figure 5. Modeling a multiplication of two 10x10 matrices using nested Stateflow `do-while` loop design patterns.



```
void DO_step(void)
{
  {
    int8_T sf_j;
    int8_T sf_i;
    int8_T sf_k;
    sf_i = 9;
    do {
      sf_j = 9;
      do {
        sf_k = 9;
        y_1[sf_i + 10 * sf_j] = 0;
        do {
          y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
            + y_1[10 * sf_j + sf_i];
          sf_k--;
        } while (sf_k >= 0);

        sf_j--;
      } while (sf_j >= 0);

      sf_i--;
    } while (sf_i >= 0);
  }
}
```

Figure 6. Generated C source code with the proper use of nested Stateflow `do-while` loop design patterns.

The reduced source code contains production code constructs, or C source code constructs in this case, that represent the algorithm more concisely. A subset of mappings from modeling design patterns to common C source code constructs can contain the following list (for other source code languages such as the C++ language, the list may contain different constructs):

- Data types, operators, and expressions such as data declarations, data type conversions, and type qualifiers
- Control flows such as if-then-else, switch, and for-loops
- Functions and program structures such as void-void functions and calling external functions
- Structures such as nested structures and bit fields
- Arrays and pointers

A set for the Simulink modeling environment is available from The MathWorks[5].

USING TARGET-OPTIMIZED CODE - During the automatic code generation process, it is efficient to replace appropriate sections of the C source code with optimized C code for a specific target. There are two techniques for doing this:

- Reuse existing handwritten or legacy code that has been tested and optimized for a specific target
- Use target-specific libraries that contain mappings of functions and operators to optimized object code

Figure 7 shows ANSI C and optimized C source code for the Infineon TriCore processor.

ANSI C code

```
void TFL_32_add_TriCore_Add_32(void)
{
  {
    int32_T tmp;
    tmp = u_1 + u_2;
    if ((u_1 < 0) && (u_2 < 0) && (tmp >= 0)) {
      tmp = MIN_int32_T;
    } else {
      if ((u_1 > 0) && (u_2 > 0) && (tmp <= 0)) {
        tmp = MAX_int32_T;
      }
    }

    y_1 = tmp;
  }
}
```

Target-optimized code for TriCore

```
inline int32_T tricore_add_s32_s32_s32_sat(int32_T a, int32_T b)
{
    return (__sat int)a + b;
}
```

Figure 7. ANSI C code and Infineon TriCore optimized code using a Target Function Library for two 32-bit fixed-point numbers.

The algorithm adds two 32-bit fixed-point numbers and performs saturation checks on the output. The second block of code is optimized using a single call to an intrinsic TriCore function that replaces the first block of code. This function is available through a Target Function Library (TFL) mapping using Real-Time Workshop Embedded Coder.

COMPILING C SOURCE CODE INTO OBJECT CODE

The previous section shows the use of modeling design patterns to optimize C source code for size. However, optimized C source code does not necessarily guarantee optimal object code in terms of size. Therefore, it is essential to understand the impact of the compilation and linking steps on the overall object code size.

Resources on embedded systems are limited. As a result, memory used to store instructions and registers used for computation are at a premium. In the matrix multiplication algorithm, execution time of the algorithm is heavily dependent on the number of instructions in the inner-most loop. We use three metrics to measure the quality of generated object code:

- Total number of instructions measured in bytes
- Number of inner-loop instructions measured in bytes
- Number of registers used

The following sections discuss three variables that affect the size of the compiled object code: microprocessor architecture, type of compiler, and compiler optimization.

CHOOSING A MICROPROCESSOR ARCHITECTURE - Figure 8 shows a Stateflow chart that implements the same matrix multiplication algorithm shown in Figure 5 using nested Stateflow for-loop design patterns.

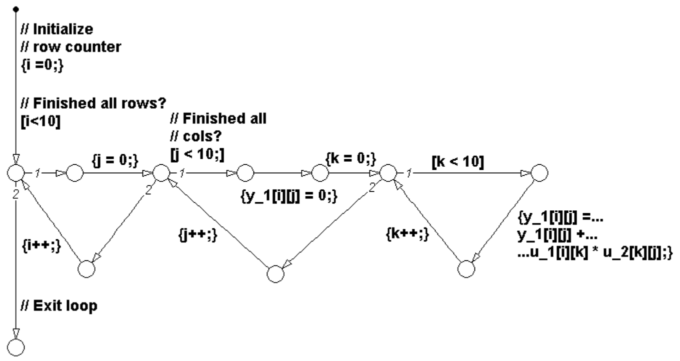


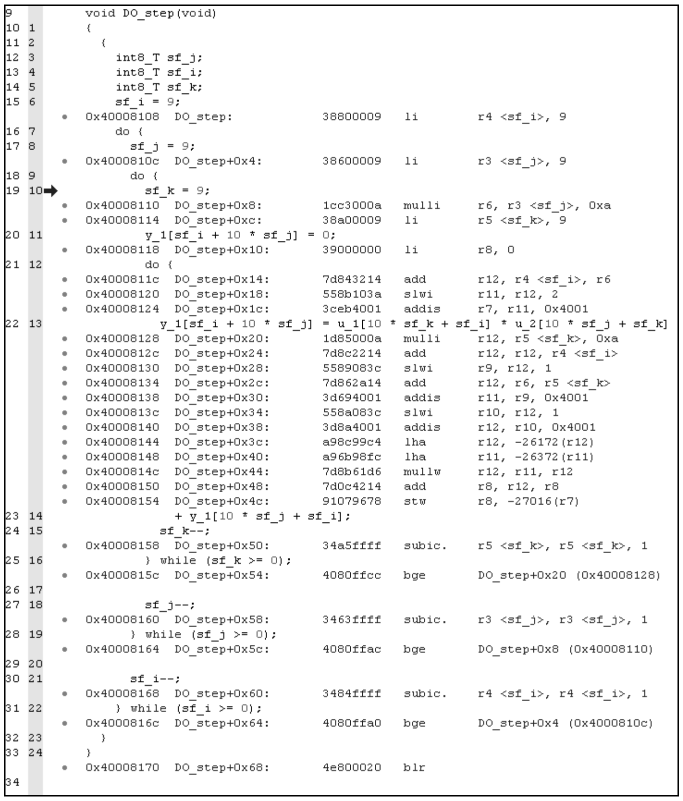
Figure 8. Modeling a multiplication of two 10x10 matrices with nested Stateflow for-loop design patterns.

The C source code generated from this model has 17 LOC, shown in Figure 9. It may appear to be more efficient compared with the 25 LOC generated using the Stateflow for-loop design pattern shown in Figure 6.

```
void FOR0_step(void)
{
  {
    int8_T sf_j;
    int8_T sf_i;
    int8_T sf_k;
    for (sf_i = 0; sf_i < 10; sf_i++) {
      for (sf_j = 0; sf_j < 10; sf_j++) {
        y_1[sf_i + 10 * sf_j] = 0;
        for (sf_k = 0; sf_k < 10; sf_k++) {
          y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
            + y_1[10 * sf_j + sf_i];
        }
      }
    }
  }
}
```

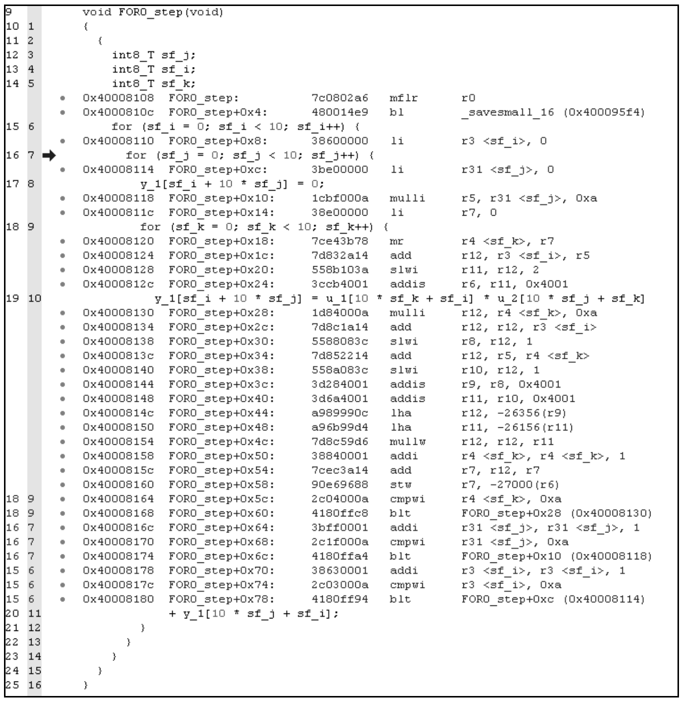
Figure 9. Generated C source code using nested Stateflow for-loop design patterns.

The C source code is compiled for the MCU 1 processor and shown in Figure 10 and Figure 11. For readability, the C source code is shown with the assembly code.



```
9            void DO_step(void)
10 1         {
11 2           {
12 3             int8_T sf_j;
13 4             int8_T sf_i;
14 5             int8_T sf_k;
15 6             sf_i = 9;
         •   0x40008108  DO_step:            38800009   li        r4 <sf_i>, 9
16 7             do {
17 8               sf_j = 9;
         •   0x4000810c  DO_step+0x4:        38600009   li        r3 <sf_j>, 9
18 9               do {
19 10                sf_k = 9;
         •   0x40008110  DO_step+0x8:        1cc3000a   mulli     r6, r3 <sf_j>, 0xa
         •   0x40008114  DO_step+0xc:        38a00009   li        r5 <sf_k>, 9
20 11                y_1[sf_i + 10 * sf_j] = 0;
         •   0x40008118  DO_step+0x10:       39000000   li        r8, 0
21 12                do {
         •   0x4000811c  DO_step+0x14:       7d843214   add       r12, r4 <sf_i>, r6
         •   0x40008120  DO_step+0x18:       558b103a   slwi      r11, r12, 2
         •   0x40008124  DO_step+0x1c:       3ceb4001   addis     r7, r11, 0x4001
22 13                  y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
         •   0x40008128  DO_step+0x20:       1d85000a   mulli     r12, r5 <sf_k>, 0xa
         •   0x4000812c  DO_step+0x24:       7d8c2214   add       r12, r12, r4 <sf_i>
         •   0x40008130  DO_step+0x28:       5589083c   slwi      r9, r12, 1
         •   0x40008134  DO_step+0x2c:       7d862a14   add       r12, r6, r5 <sf_k>
         •   0x40008138  DO_step+0x30:       3d694001   addis     r11, r9, 0x4001
         •   0x4000813c  DO_step+0x34:       558a083c   slwi      r10, r12, 1
         •   0x40008140  DO_step+0x38:       3d8a4001   addis     r12, r10, 0x4001
         •   0x40008144  DO_step+0x3c:       a98c99c4   lha       r12, -26172(r12)
         •   0x40008148  DO_step+0x40:       a96b98fc   lha       r11, -26372(r11)
         •   0x4000814c  DO_step+0x44:       7d8b61d6   mullw     r12, r11, r12
         •   0x40008150  DO_step+0x48:       7d0c4214   add       r8, r12, r8
         •   0x40008154  DO_step+0x4c:       91079678   stw       r8, -27016(r7)
23 14                   + y_1[10 * sf_j + sf_i];
24 15                 sf_k--;
         •   0x40008158  DO_step+0x50:       34a5ffff   subic.    r5 <sf_k>, r5 <sf_k>, 1
25 16              } while (sf_k >= 0);
         •   0x4000815c  DO_step+0x54:       4080ffcc   bge       DO_step+0x20 (0x40008128)
26 17
27 18              sf_j--;
         •   0x40008160  DO_step+0x58:       3463ffff   subic.    r3 <sf_j>, r3 <sf_j>, 1
28 19            } while (sf_j >= 0);
         •   0x40008164  DO_step+0x5c:       4080ffac   bge       DO_step+0x8 (0x40008110)
29 20
30 21            sf_i--;
         •   0x40008168  DO_step+0x60:       3484ffff   subic.    r4 <sf_i>, r4 <sf_i>, 1
31 22          } while (sf_i >= 0);
         •   0x4000816c  DO_step+0x64:       4080ffa0   bge       DO_step+0x4 (0x4000810c)
32 23          }
33 24        }
         •   0x40008170  DO_step+0x68:       4e800020   blr
34
```

Figure 10. C source and assembly codes for the matrix multiplication algorithm using `do-while` loop design pattern on MCU 1.



```
9            void FOR0_step(void)
10 1         {
11 2           {
12 3             int8_T sf_j;
13 4             int8_T sf_i;
14 5             int8_T sf_k;
         •   0x40008108  FOR0_step:          7c0802a6   mflr      r0
         •   0x4000810c  FOR0_step+0x4:      480014e9   bl        _savesmall_16 (0x400095f4)
15 6             for (sf_i = 0; sf_i < 10; sf_i++) {
         •   0x40008110  FOR0_step+0x8:      38600000   li        r3 <sf_i>, 0
16 7               for (sf_j = 0; sf_j < 10; sf_j++) {
         •   0x40008114  FOR0_step+0xc:      3be00000   li        r31 <sf_j>, 0
17 8                 y_1[sf_i + 10 * sf_j] = 0;
         •   0x40008118  FOR0_step+0x10:     1cbf000a   mulli     r5, r31 <sf_j>, 0xa
         •   0x4000811c  FOR0_step+0x14:     38e00000   li        r7, 0
18 9                 for (sf_k = 0; sf_k < 10; sf_k++) {
         •   0x40008120  FOR0_step+0x18:     7ce43b78   mr        r4 <sf_k>, r7
         •   0x40008124  FOR0_step+0x1c:     7d832a14   add       r12, r3 <sf_i>, r5
         •   0x40008128  FOR0_step+0x20:     558b103a   slwi      r11, r12, 2
         •   0x4000812c  FOR0_step+0x24:     3ccb4001   addis     r6, r11, 0x4001
19 10                  y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
         •   0x40008130  FOR0_step+0x28:     1d84000a   mulli     r12, r4 <sf_k>, 0xa
         •   0x40008134  FOR0_step+0x2c:     7d8c1a14   add       r12, r12, r3 <sf_i>
         •   0x40008138  FOR0_step+0x30:     5588083c   slwi      r8, r12, 1
         •   0x4000813c  FOR0_step+0x34:     7d852214   add       r12, r5, r4 <sf_k>
         •   0x40008140  FOR0_step+0x38:     558a083c   slwi      r10, r12, 1
         •   0x40008144  FOR0_step+0x3c:     3d284001   addis     r9, r8, 0x4001
         •   0x40008148  FOR0_step+0x40:     3d6a4001   addis     r11, r10, 0x4001
         •   0x4000814c  FOR0_step+0x44:     a989990c   lha       r12, -26356(r9)
         •   0x40008150  FOR0_step+0x48:     a96b99d4   lha       r11, -26156(r11)
         •   0x40008154  FOR0_step+0x4c:     7d8c59d6   mullw     r12, r12, r11
         •   0x40008158  FOR0_step+0x50:     38840001   addi      r4 <sf_k>, r4 <sf_k>, 1
         •   0x4000815c  FOR0_step+0x54:     7cec3a14   add       r7, r12, r7
         •   0x40008160  FOR0_step+0x58:     90e69688   stw       r7, -27000(r6)
18 9     •   0x40008164  FOR0_step+0x5c:     2c04000a   cmpwi     r4 <sf_k>, 0xa
18 9     •   0x40008168  FOR0_step+0x60:     4180ffc8   blt       FOR0_step+0x28 (0x40008130)
16 7     •   0x4000816c  FOR0_step+0x64:     3bff0001   addi      r31 <sf_j>, r31 <sf_j>, 1
16 7     •   0x40008170  FOR0_step+0x68:     2c1f000a   cmpwi     r31 <sf_j>, 0xa
16 7     •   0x40008174  FOR0_step+0x6c:     4180ffa4   blt       FOR0_step+0x10 (0x40008118)
15 6     •   0x40008178  FOR0_step+0x70:     38630001   addi      r3 <sf_i>, r3 <sf_i>, 1
15 6     •   0x4000817c  FOR0_step+0x74:     2c03000a   cmpwi     r3 <sf_i>, 0xa
15 6     •   0x40008180  FOR0_step+0x78:     4180ff94   blt       FOR0_step+0xc (0x40008114)
20 11                  + y_1[10 * sf_j + sf_i];
21 12                }
22 13              }
23 14            }
24 15          }
25 16        }
```

Figure 11. C source and assembly codes for matrix multiplication using for-loop design pattern on MCU 1.

Table 1 shows the metrics for the quality of the generated object code on several different processors. The use of a `do-while` loop design pattern instead of a for-loop design pattern results in notably better metrics for the algorithm on the MCU 1 processor. The results are less significant for the other two processors. This shows an example where measuring LOC at the C source code level is insufficient to gauge the quality of the object code; architecture dependency may affect the choice of modeling design pattern used in the model.

Design patterns	Total number of instructions (bytes)	Inner-loop instructions (bytes)	Number of registers
MCU 1			
for-loop	120	56	12
`do-while` loop	104	52	10
MCU 2			
for-loop	78	38	7
`do-while` loop	74	36	7
DSP 2			
for-loop	114	52	8
`do-while` loop	104	48	8

Table 1. Metrics for quality of the generated object code for the matrix multiplication algorithm on different processors.

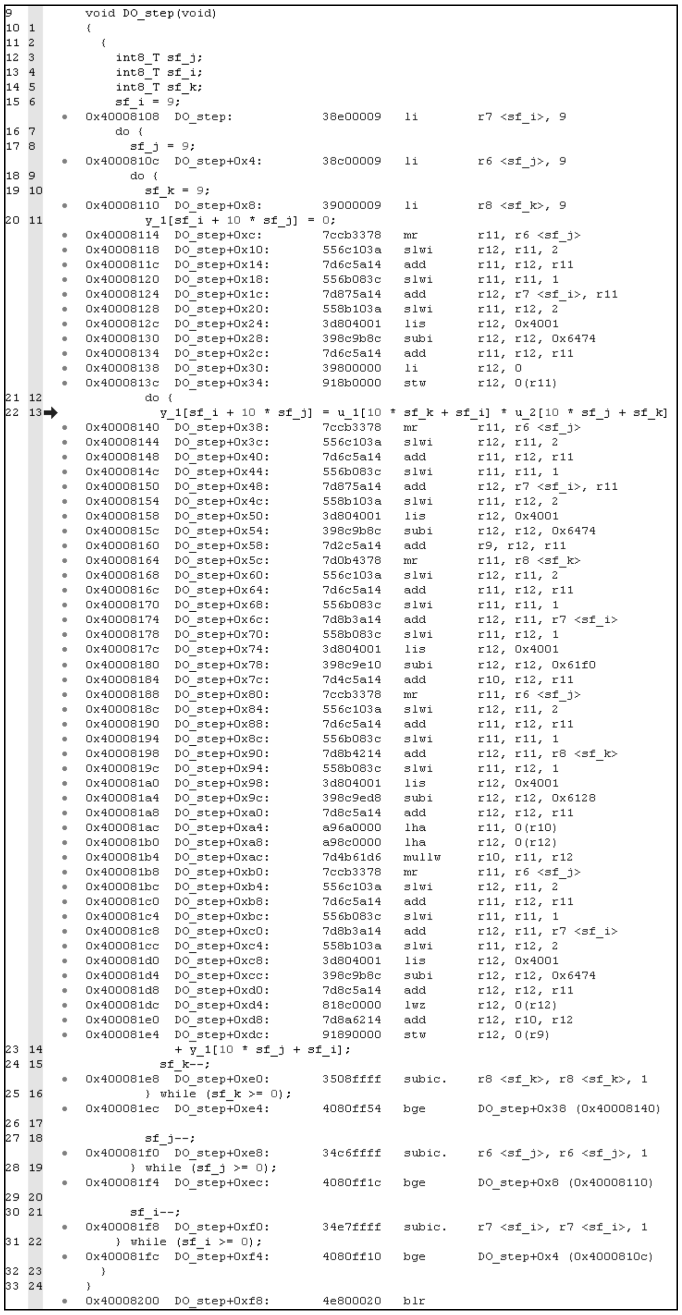
CHOOSING A COMPILER - The intrinsic characteristics of a compiler can affect the quality of the object code. Table 2 summarizes the metrics for object code generated for the for-loop design pattern example using the `-Osize` compiler optimization flag available for both compilers.

Compiler	Total number of instructions (bytes)
MCU 1	
Green Hills	120
GCC	204
MCU 2	
Green Hills	78
GCC	118
DSP 1	
Green Hills	128
GCC	220

Table 2. Metrics for quality of the generated object code for the matrix multiplication algorithm on different compilers and processors.

USING COMPILER-SPECIFIC OPTIMIZATIONS - The `–Osize` optimization flag minimizes object code size Using the compiler's code optimization technologies. It may optimize code size at the expense of speed. A similar flag, `–O`, can also be used to optimize object code for a balance of both size and speed.

Figure 10 shows the assembly code generated from the `do-while` loop design pattern with the `–Osize` optimization flag for the MCU 1 processor. Compared with this result, the assembly code is larger without the use of the flag, as shown in Figure 12.



```
     void DO_step(void)
     {
       {
         int8_T sf_j;
         int8_T sf_i;
         int8_T sf_k;
         sf_i = 9;
  •  0x40008108  DO_step:            38e00009   li         r7 <sf_i>, 9
         do {
           sf_j = 9;
  •  0x4000810c  DO_step+0x4:        38c00009   li         r6 <sf_j>, 9
           do {
             sf_k = 9;
  •  0x40008110  DO_step+0x8:        39000009   li         r8 <sf_k>, 9
             y_1[sf_i + 10 * sf_j] = 0;
  •  0x40008114  DO_step+0xc:        7ccb3378   mr         r11, r6 <sf_j>
  •  0x40008118  DO_step+0x10:       556c103a   slwi       r12, r11, 2
  •  0x4000811c  DO_step+0x14:       7d6c5a14   add        r11, r12, r11
  •  0x40008120  DO_step+0x18:       556b083c   slwi       r11, r11, 1
  •  0x40008124  DO_step+0x1c:       7d875a14   add        r12, r7 <sf_i>, r11
  •  0x40008128  DO_step+0x20:       558b103a   slwi       r11, r12, 2
  •  0x4000812c  DO_step+0x24:       3d804001   lis        r12, 0x4001
  •  0x40008130  DO_step+0x28:       398c9b8c   subi       r12, r12, 0x6474
  •  0x40008134  DO_step+0x2c:       7d6c5a14   add        r11, r12, r11
  •  0x40008138  DO_step+0x30:       39800000   li         r12, 0
  •  0x4000813c  DO_step+0x34:       918b0000   stw        r12, 0(r11)
             do {
               y_1[sf_i + 10 * sf_j] = u_1[10 * sf_k + sf_i] * u_2[10 * sf_j + sf_k]
  •  0x40008140  DO_step+0x38:       7ccb3378   mr         r11, r6 <sf_j>
  •  0x40008144  DO_step+0x3c:       556c103a   slwi       r12, r11, 2
  •  0x40008148  DO_step+0x40:       7d6c5a14   add        r11, r12, r11
  •  0x4000814c  DO_step+0x44:       556b083c   slwi       r11, r11, 1
  •  0x40008150  DO_step+0x48:       7d875a14   add        r12, r7 <sf_i>, r11
  •  0x40008154  DO_step+0x4c:       558b103a   slwi       r11, r12, 2
  •  0x40008158  DO_step+0x50:       3d804001   lis        r12, 0x4001
  •  0x4000815c  DO_step+0x54:       398c9b8c   subi       r12, r12, 0x6474
  •  0x40008160  DO_step+0x58:       7d2c5a14   add        r9, r12, r11
  •  0x40008164  DO_step+0x5c:       7d0b4378   mr         r11, r8 <sf_k>
  •  0x40008168  DO_step+0x60:       556c103a   slwi       r12, r11, 2
  •  0x4000816c  DO_step+0x64:       7d6c5a14   add        r11, r12, r11
  •  0x40008170  DO_step+0x68:       556b083c   slwi       r11, r11, 1
  •  0x40008174  DO_step+0x6c:       7d8b3a14   add        r12, r11, r7 <sf_i>
  •  0x40008178  DO_step+0x70:       558b083c   slwi       r11, r12, 1
  •  0x4000817c  DO_step+0x74:       3d804001   lis        r12, 0x4001
  •  0x40008180  DO_step+0x78:       398c9e10   subi       r12, r12, 0x61f0
  •  0x40008184  DO_step+0x7c:       7d4c5a14   add        r10, r12, r11
  •  0x40008188  DO_step+0x80:       7ccb3378   mr         r11, r6 <sf_j>
  •  0x4000818c  DO_step+0x84:       556c103a   slwi       r12, r11, 2
  •  0x40008190  DO_step+0x88:       7d6c5a14   add        r11, r12, r11
  •  0x40008194  DO_step+0x8c:       556b083c   slwi       r11, r11, 1
  •  0x40008198  DO_step+0x90:       7d8b4214   add        r12, r11, r8 <sf_k>
  •  0x4000819c  DO_step+0x94:       558b083c   slwi       r11, r12, 1
  •  0x400081a0  DO_step+0x98:       3d804001   lis        r12, 0x4001
  •  0x400081a4  DO_step+0x9c:       398c9ed8   subi       r12, r12, 0x6128
  •  0x400081a8  DO_step+0xa0:       7d8c5a14   add        r12, r12, r11
  •  0x400081ac  DO_step+0xa4:       a96a0000   lha        r11, 0(r10)
  •  0x400081b0  DO_step+0xa8:       a98c0000   lha        r12, 0(r12)
  •  0x400081b4  DO_step+0xac:       7d4b61d6   mullw      r10, r11, r12
  •  0x400081b8  DO_step+0xb0:       7ccb3378   mr         r11, r6 <sf_j>
  •  0x400081bc  DO_step+0xb4:       556c103a   slwi       r12, r11, 2
  •  0x400081c0  DO_step+0xb8:       7d6c5a14   add        r11, r12, r11
  •  0x400081c4  DO_step+0xbc:       556b083c   slwi       r11, r11, 1
  •  0x400081c8  DO_step+0xc0:       7d8b3a14   add        r12, r11, r7 <sf_i>
  •  0x400081cc  DO_step+0xc4:       558b103a   slwi       r11, r12, 2
  •  0x400081d0  DO_step+0xc8:       3d804001   lis        r12, 0x4001
  •  0x400081d4  DO_step+0xcc:       398c9b8c   subi       r12, r12, 0x6474
  •  0x400081d8  DO_step+0xd0:       7d8c5a14   add        r12, r12, r11
  •  0x400081dc  DO_step+0xd4:       818c0000   lwz        r12, 0(r12)
  •  0x400081e0  DO_step+0xd8:       7d8a6214   add        r12, r10, r12
  •  0x400081e4  DO_step+0xdc:       91890000   stw        r12, 0(r9)
                 + y_1[10 * sf_j + sf_i];
               sf_k--;
  •  0x400081e8  DO_step+0xe0:       3508ffff   subic.     r8 <sf_k>, r8 <sf_k>, 1
             } while (sf_k >= 0);
  •  0x400081ec  DO_step+0xe4:       4080ff54   bge        DO_step+0x38 (0x40008140)

             sf_j--;
  •  0x400081f0  DO_step+0xe8:       34c6ffff   subic.     r6 <sf_j>, r6 <sf_j>, 1
           } while (sf_j >= 0);
  •  0x400081f4  DO_step+0xec:       4080ff1c   bge        DO_step+0x8 (0x40008110)

           sf_i--;
  •  0x400081f8  DO_step+0xf0:       34e7ffff   subic.     r7 <sf_i>, r7 <sf_i>, 1
         } while (sf_i >= 0);
  •  0x400081fc  DO_step+0xf4:       4080ff10   bge        DO_step+0x4 (0x4000810c)
       }
     }
  •  0x40008200  DO_step+0xf8:       4e800020   blr
```

Figure 12. Assembly code generated without compiler optimization `–Osize` flag for MCU 1.

A summary of the comparisons is shown in Table 3. As expected, compiler optimizations affect the quality of the object code[6].

`-Osize` optimization flag	Total number of instructions (bytes)	Inner-loop instructions (bytes)	Number of registers
On	104	52	10
Off	248	172	7

Table 3. Metrics for quality of the generated object code for the matrix multiplication algorithm on MCU 1

CONCLUSION

In Model-Based Design, the quality of the object code measured in terms of size is critical for final deployment. It may impact pricing decisions such as the choice of processor and memory. Thus, incorporating techniques to optimize object code as a goal of embedded controller design can significantly reduce costs and development times.

Several key steps in the code generation workflow impact the size of the final object code. Techniques such as using modeling patterns and target-optimized code can streamline the generated C source code. Furthermore, the choice of microprocessor architecture, compiler, and compiler-specific optimizations can affect the choice of the modeling pattern.

To gain maximum leverage from these techniques, organizations can invest in:

- Undertaking detailed studies to gain a better understanding of various parameters that optimize object code at these steps in the workflow
- Establishing a culture that proactively data mines and gathers these optimizations into a set of best practices that serve as organizational memory for future designs

ACKNOWLEDGMENTS

The authors would like to acknowledge the following fellow MathWorks staff who contributed to the development of the ideas used in this paper and the writing of this paper: Arvind Jayaraman, Scott Prochaska, Thomas Maier-Komor, and Tom Erkkinen.

REFERENCES

1. P. F. Smith, S. Prabhu, J. Friedman, Best Practices for Establishing a Model-Based Design Culture, SAE Paper 2007-01-0777, 2007.
2. T. Erkkinen, S. Breiner, Automatic Code Generation—Technology Adoption Lessons Learned from Commercial Vehicle Case Studies, SAE Paper 2007-01-4249, 2007.
3. J. L. Hennessy, D. A. Patterson, D. Goldberg, Computer Architecture: A Quantitative Approach, 3rd Edition, Morgan Kaufmann Publishers Inc., 2002.
4. E. Freeman, E. Freeman, B. Bates, K. Sierra, Head First Design Patterns, 1st Edition, O'Reilly Media, Inc., 2004.
5. The MathWorks, Inc., Modeling Patterns for C Constructs: www.mathworks.com/support/solutions/data/1-6AWSQ9.html.
6. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, 2006 Addison-Wesley.

CONTACT

Bill Chou, embedded code generation and verification marketing, The MathWorks, Bill.Chou@mathworks.com

Saurabh Mahapatra, Simulink platform marketing, The MathWorks, Saurabh.Mahapatra@mathworks.com