Techniques for Generating and Measuring Production Code Constructs from Controller Models

Bill Chou, Saurabh Mahapatra
The MathWorks

Copyright © 2009 The MathWorks. Published by SAE International with permission.

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:
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.](image)

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.](image)

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.

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 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.
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_l[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.

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

```c
void TFL32_add_TriCore_Add32(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

```c
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.

```
# Initialize
// row counter
(t = 0)

# Finished all rows?
[t+10]

# Finished all cols?
[k + 10]

<<[y10][0] = 0;>>
<<[y10][0] = ...
   ...[u2][0][4] + [u2][0][10]>>

(j = 0;)

<<[j][j][0] = 0;>>
<<[j][j][0] = ...
   ...[u2][0][4] + [u2][0][10]>>

// Exit loop
```

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 FPLD_step(void)
{
    ...
}
```

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

Design patterns | Total number of instructions (bytes) | Inner-loop instructions (bytes) | Number of registers
--- | --- | --- | ---
MCU 1 | 120 | 56 | 12
-MCU 2 | 78 | 38 | 7
-DSP | 114 | 52 | 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.

Using compiler-specific optimizations - The -Osize optimization flag minimizes object code size by 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.

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.
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.

<table>
<thead>
<tr>
<th>-Osize optimization flag</th>
<th>Total number of instructions (bytes)</th>
<th>Inner-loop instructions (bytes)</th>
<th>Number of registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>On</td>
<td>104</td>
<td>52</td>
<td>10</td>
</tr>
<tr>
<td>Off</td>
<td>248</td>
<td>172</td>
<td>7</td>
</tr>
</tbody>
</table>

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 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

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

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.