Comparative Study of Fixed-Point and Floating-Point Code for a Fixed-Point Micro

Anoop C V, Betta C
Delphi Technical Center, India.

ABSTRACT

Model-based development relies a great deal on the code generation tool. TargetLink® fills this space with a number of features allowing efficient production-code generation. Traditionally in the automotive industry, high-volume ECUs demand low-cost fixed-point microcontroller units. Use of fixed-point processors usually warrants the use of fixed-point autocoding which is effort-intensive for highly math-intensive algorithms. This paper focuses on the possibility of harnessing the ease of floating-point autocoding with TargetLink® in conjunction with compiler libraries. These libraries provide a set of procedure calls to do floating-point arithmetic on fixed-point processors and have very efficient implementation that leads to a good performance, making this option very practical and reducing the autocoding effort. This effectively means shorter product development time without sacrificing product quality and robustness. The paper focuses on a study based on our experience involving comparison of fixed-point and floating-point code performance for processors with no floating-point hardware support.

INTRODUCTION

The automotive industry constantly works towards finding and using robust and economical solutions to apply technology and implement features driven by control algorithms. With time-to-market being important in the industry, reducing the same without sacrificing the quality of products has pushed this industry to look at various development methodologies. Adopting Model-Based Development (MBD) along with cost-effective hardware for running the algorithms has definitely been a move in this direction of having reduced time-to-market without compromising on product quality.

Fixed-point processors provide for cost-effective hardware compared to floating-point processors. Floating-point processors are more expensive than fixed-point integer processors due the complex internal circuitry and the increased memory requirements. As the volume for electronic control units (ECU) needed for production is high, customers prefer to use fixed-point processors. However, this comes at the expense of higher cost for the implementation of algorithms in fixed-point. The engineering effort for floating-point autocoding and testing is less compared to the fixed-point autocoding and testing. Apart from effort, algorithm performance is also better in floating-point implementation compared to fixed-point implementation.

The ideal situation would therefore be the ability to use fixed-point processors, while generating the autocode as easily as with floating-point autocoding process.

MOTIVATION

Lev Vitkin et al. in [1] clearly describes the process of selection between fixed-point and floating-point processors. The authors propose the use of fixed-point autocoding for fixed-point processors and floating-point autocoding for floating-point processors while comparing the efforts for the same. The authors go on to suggest that almost all stages are much simpler to perform and require less engineering effort for floating-point autocoding than fixed-point autocoding. The effort reduction they describe arises from the elimination of setting some autocode attributes and not having to perform the verifications that are related to fixed-point math implementation.

The formula for Model Complexity for the estimation of autocoding time is discussed in [2]. Here the adjustment of coefficients based upon the type of auto-generated code is estimated. The adjusting coefficients estimated by the authors
have a significant higher value for fixed-point when compared to the floating-point, showing how effort required for fixed-point autocode is much higher compared to that for floating-point.

Also, unit-testing needs to be performed for only fixed-point autocode because there is no need to check for overflow and/or underflow for floating-point autocode. Chandrashekar, MS et al. [3], discuss that the Model-based Software Development process scores higher than conventional hand-code process with respect to development time and software quality and the engineering effort, and for fixed-point autocoding, the effort is approximately 50% less when compared to conventional hand-coding. The authors discuss that the effort reduction should be even higher than 50% for floating-point autocode as the fixed-point math analysis and unit-testing steps would be eliminated.

Also, in [4], the authors Lev Vitkin and TK Jestin discuss that when memory and throughput limitations are relaxed and processor power is increased to accommodate floating-point math, the process of producing autocode will be much closer to the ideal “push-button” solution.

MODEL-BASED SOFTWARE DEVELOPMENT

Model-based software development (MBSD) is divided mainly into four phases: Model based design, Model validation, Implementation and Testing. Model-based design starts with requirements being modeled to form the design. Model validation can be done in the PC-based environment by simulating the model and testing for functionality. A rapid prototype feature may also be used for complete validation of the model. Implementation usually involves setting different code-generation properties for various blocks of the model and using different tools like TargetLink® for code generation. A considerable amount of effort reduction in the implementation phase is achieved by using autocoding tools. Testing the auto-generated code involves testing the code against the model by verifying code outputs to match model outputs.

AUTOCODING

There are several steps to be followed in autocoding. The model developed by the system engineers for autotocoding should be well reviewed by the software engineers. Model guidelines for the autocode are designed in such a way to ensure that model is autocodable and will generate efficient code for the production software.

The model to be used for code generation should be compatible with tools used for code generation. When TargetLink® is used for the code-generation process the Simulink® model is converted/ auto-prepared into TargetLink® model. The interface variables and types are defined based on the ranges and resolution information provided. Interface variables are the input and output variables of each component module in the integrated model. The class, data type and variable name etc., are to be well defined for interface variables as well as intermediate variables. The variable’s details like class, etc., are declared in the same autocoded module and datatype calculation done based on the system range and resolution defined by the system engineers. This is an example of a Main Heading section. This section will include sub-sections.

The process of autocode generation may be classified in one of two types:

FIXED-POINT AUTOCODING - Integer processors are less expensive than floating-point processors. The cost benefit is one of the key reasons to make use of a fixed-point controller. In fixed-point autocoding, the achievable range and resolution must be determined based on the range and resolution of the inputs to algorithm so that appropriate data type can be chosen. To implement intense math algorithms, a significant amount of time must be spent on fixed-point analysis to determine how many bits should be used for range and how many bits should be used for resolution.

Range calculation starts from the beginning of the data flow in the model and then propagates its ranges to the end of the data flow. The results of each range calculation determine the number of bits needed for the integer part calculation. These range calculations are performed for each math operation in the model. Range loss of any blocks may lead to functional error. Resolution analysis starts with system requirements of the resolution of the output signal. This resolution prediction, similar to range estimation, must be repeated for each math operation. The resolution estimation starts from the end of the data flow in the model and propagates to the beginning of the data flow of the model. The results of the resolution analysis determine the number of bits required for the fractional part of the variable. Based on the range and the resolution analysis, the next step is set appropriate data type for that variable with class, variable name, etc.
Testing of the fixed-point autocode is an important phase to catch any functional errors that may occur due to overflow / underflow. Unit testing is used to verify that the model and the code outputs behave the same. A Unit is a smallest building block in the model, such as Simulink® subsystem, that does not include another subsystem in the model. The purpose of Unit testing is to cover the worst case range and resolution based on wider inputs (ranges and resolution) and calibrations. This testing helps to verify the overflow and underflow check of the data type.

FLOATING-POINT AUTOCODING - Floating-point autocoding steps are quite similar to fixed-point autocoding. Here the software properties to be manually and individually set are significantly less than those of fixed-point autocoding. It requires less effort, as little or no effort needs to be spent to calculate range and resolution as well as to perform overflow or underflow checks, as in practice, the range and resolution supported by floating-point representation is sufficient for most algorithms used in automotive applications. Production-code review will also be simpler in case of floating-point autocode for the above mentioned reasons. Also, since floating point data types cover wide range and resolution, traditional unit testing can be skipped.

THE PROPOSED APPROACH

The paper proposes using floating-point autocoding for fixed-point processors. In doing so, one leverages the ease of autocoding with tools like TargetLink® while also taking advantage of the low cost of the fixed-point processors.

Here we consider a system with no floating-point hardware.

Compiling for a system with a hardware coprocessor for floating-point support can be done with appropriate compiler options. The compiler usually makes use of the hardware coprocessor. Programs compiled to use a floating-point coprocessor perform basic floating-point arithmetic operations using floating-point machine instructions for the target coprocessor. Compilers also provide compile options for a system without a coprocessor. In this case, floating-point arithmetic operations need to be implemented in software. When programs are compiled to use software floating-point for systems with no floating-point instruction set available, certain libraries must provide a set of procedure calls to do floating-point arithmetic. These procedures are in the software floating-point library, and the floating-point arguments are passed and returned in integer registers.

Often, the floating-point libraries are fast because they are written in assembly language and specifically optimized for each processor. Performance using the floating-point libraries may be comparable or even better than the code implementation in fixed-point. This point brings up the next possibility of actually implementing the floating-point autocoding and saving significant effort. TargetLink® provides for easy configuration and code-generation when using floating-point autocoding.

The study compares a section of the safety algorithm with considerable computation autoded using traditional fixed-point math against the same algorithm autocoded in floating-point. The algorithm was run on the same set of test Vectors and it was found that the floating-point implementation was performed better.

RESULTS

Table below [Table1] provides the execution time of a safety algorithm that is averaged over multiple runs with different inputs. The algorithm was selected to do the study because it had a considerable amount of math blocks to be implemented. The algorithm implemented as a Simulink® model was autoded using TargetLink®. The hardware did not have any support for floating-point execution.

First, the model was autoded using TargetLink® to generate fixed-point code. The code was then compiled using the RealView® compilation tools for ARM® core. The module was run on the hardware with certain set of test-vectors provided as inputs. The execution time was measured for every cycle and averaged to obtain the results in Table1.
Next, as per the proposed approach, the same model was autocoded to generate floating-point code using TargetLink®, and the code was compiled using the same tool-chain. The module was run on the same hardware with same set of test-vectors as previously provided. This time the code was compiled and linked with the floating-point library. The execution time was measured for every cycle and averaged to obtain the results also in Table 1.

<table>
<thead>
<tr>
<th>Fixed-Point implementation</th>
<th>Floating-Point implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Execution Time (micro second)</td>
<td>Average Execution Time (micro second)</td>
</tr>
<tr>
<td>37.38312452</td>
<td>25.2764924</td>
</tr>
</tbody>
</table>

Table 1: Results comparing the average execution time for the selected algorithm autocoded in fixed-point against the algorithm autocoded in floating-point

Table 1 shows that floating-point autocoding when compiled with a floating-point library executes as much as 32.3% better than the fixed-point implementation on hardware with no support for floating-point execution.

CONCLUSION

As a result of executing our proposed approach, we see that the floating-point auto-generated code when compiled with a floating-point library executed as much as 32.3% faster than the fixed-point implementation. The decision not to use fixed-point autocoding while using a fixed-point processor should be carefully evaluated only after having done a complete study of the amount of impact or advantage gained from floating-point implementation.

This study offers an option for using floating-point autocoding for fixed-point processors. The results described above are encouraging enough and show a considerable performance improvement. This improvement is in addition to the reduction in effort required by floating-point autocoding that TargetLink® provides.

ACKNOWLEDGMENTS

The authors would like to acknowledge and thank the teams in Delphi Technical Center India, Kokomo and China for their contributions and guidance.

REFERENCES

CONTACT

Anoop C V
e-mail: anoop.cv@delphi.com,
phone: 080-30777608

Betta C
e-mail: betta.shageesh@delphi.com,
phone: 080-30777602