Proceedings of Informing Science & IT Education Conference (InSITE) 2014

Cite as: Stolze, T. & Kramer, K-D. (2014). Runtime optimization of generated code *Proceedings of Informing Science & IT Education Conference (InSITE) 2014* (pp. 313-322). Retrieved from http://Proceedings.InformingScience.org/InSITE2014/InSITE14p313-311Stolze0698.pdf

Runtime Optimization of Generated Code

Thomas Stolze and Klaus-Dietrich Kramer
Department of Automation and Computer Science, Harz University, Wernigerode, Germany

tstolze@hs-harz.de kkramer@hs-harz.de

Abstract

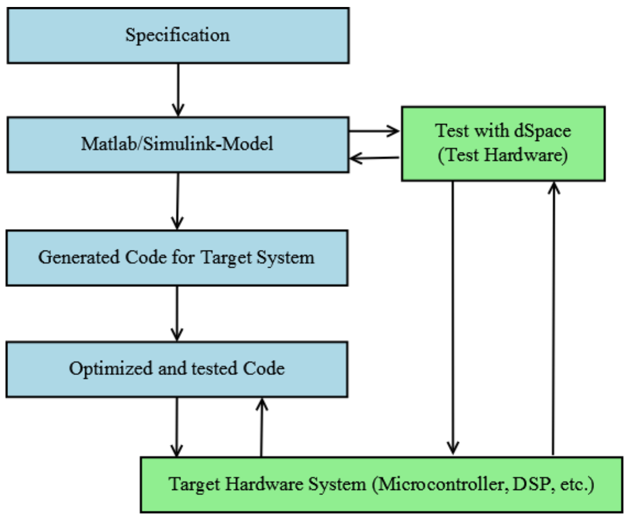
CASE-Tools are state-of-the-art when code has to be generated out of a software model. They offer high flexibility and sophisticated usability. Time savings in software development are huge compared to traditional programming techniques, and lower production times lead to lower costs. Many applications require real time processing. But real time demands are quite difficult to handle with CASE tools. There are too many constraints and only a limited number of them can be processed by the tools. Therefore, the following approaches deal with runtime optimization of generated code. Starting with the current situation successive methods how to optimize code for real time requirements are shown.

Keywords: code generation, Matlab/Simulink, code optimization, real time, performance

Introduction / Motivation

The software engineering process is shaped by so-called CASE-Tools (Computer Aided Software Engineering Tools), especially in order to program Embedded Systems. Among many other software products Matlab/Simulink (Mathworks, 1997) in combination with dSPACE (dSPACE GmbH, 2003a, 2003b) is one possible tool chain. These CASE-Tools support engineers while developing their products by providing flexible possibilities to design, to simulate and to test systems. When tested successfully the tools help to port the code to an underlying embedded hardware device. Figure 1 illustrates typical steps when developing Software with CASE tools – here with Matlab/Simulink and dSpace. But the figure should be seen as a general procedure. The hardware interactions of this development process are marked green (right side / bottom) while the software development is printed in blue color (left side). The distinctiveness of this procedure is the early interaction of the Simulink model and the dSpace hardware system for testing purposes. With the help of dSPACE simulated results can be tested with the real hardware, even before the first line of code is generated for the target hardware system, e.g., a microcontroller.

The resulting high degree of flexibility is appreciated by engineers. They are able to run simulations and tests with real hardware like dSPACE systems. But this flexibility often rivals with real time demands when trying to generate runtime-optimized code out of a simulation model. Because of the automated approach generating code for certain hardware architecture and optimizing its runtime at the same time is limited. The huge amount of constraints makes it impossible to transfer all information to the CASE tools and to let the tools process them automatically. So, if real time restrictions cannot be met by automatically generated code, a manual optimization is necessary.


Specification
Matlab/Simulink-Model
Test with dSpace (Test Hardware)
Generated Code for Target System
Optimized and tested Code
Target Hardware System (Microcontroller, DSP, etc.)

Figire 1: Typical Software Development Process with Matlab/Simulink and dSpace

By means of the project "Chainless Bike" an optimization strategy for generated code can be presented which can be adapted to other projects as well. In the example the bike is driven electrically, but behaves like a classic bike. The electric power provided by a generator and an accumulator is used by an electric motor at the rear for accelerating the bike. Because of the lack of the classic chain a microcontroller can perfectly use all the resulting degrees of freedom when riding the bike. Recuperation of braking energy is possible as well as stepless shifting. It is a product that uses the software development process given in Figure 1. The Figure 2 shows a prototype of the bike. It can be operated by a smart phone attached to the handlebars which communicates with a microcontroller.

The project makes use of the tools Matlab/Simulink and the hardware dSPACE. For controlling purposes (motor and generator control, user interaction, battery management) a microcontroller from Infineon's XC2000 family is utilized. It provides suitable peripherals and is available as a target controller in Matlab/Simulink which means code can be generated for it. To transfer the code to its machine-executable form the Integrated Development Environment (IDE) Tasking VX Toolset is used. To support the currents needed for riding the bike a special power amp circuit is connected to the microcontroller.

Figure 2: Chainless Bike prototype (X-PESA)

Runtime Problems and Optimization Approaches

Methodology and Preparations

Starting with the Simulink model of the bike the so-called Real-Time Workshop, a tool from Simulink, is able to transform the model into C code for the microcontroller. Therefore, C code is generated from the blocks within the Simulink model. With the help of Tasking VX Toolset it can be loaded onto the microcontroller. But the generated code lacks of execution speed. That is why the sampling frequency for a stable control cannot be met, and the control does not work on the microcontroller until optimized. In case of the bike an optimization by a factor greater than 10 is required in order to get the cycle time from several milliseconds to a cycle time of less than 100 µs. A sampling frequency of at least 10 kHz has to be achieved.

This problem also affects other projects which require certain sample times and make use of generated code. Generally speaking, the shorter the execution time of a control algorithm, the better the quality and stability of the control. For solving this problem an exact analysis of the algorithm runtimes and a classification of potentials of optimization are required. For a decision whether or not to optimize certain parts of the model the cost of a particular optimization is crucial. The optimization should only be implemented if the optimization potential in relation to the costs for the required changes is high enough. Therefore, the execution times of the parts of the control algorithms have to be determined. When using a microcontroller this can easily be achieved by using an oscilloscope and toggling a port pin. Using timers for measurement is not recommended since necessary interrupt service routines may negatively affect the runtime. A big support is the Simulink report of the code generation which shows the Simulink blocks and their matching C code counterparts.

Model Optimization

A general consideration has to be made which elements of the CASE tool model are essential for the control and affect the sampling time on the microcontroller. All subsidiary elements (e.g., only implemented for testing and debugging) have to be deleted. Then the model is in an optimized state. In conclusion, no unnecessary code can be generated. Especially in early project stages several elements are often calculated in parallel because of testing purposes to determine which solution is the best one. So all but the best one can be deleted saving execution time by executing only one code path.

Calculation Precision and Instruction Set

By knowing the characteristics of the preferred microcontroller some optimization potentials in the CASE tool can be identified. In case of Simulink blocks using floating-point calculation may be investigated to run with fixed-point arithmetic instead. It is not only the execution speed what counts, but also the precision of the results. But the execution of floating-point code on a fixed-point microcontroller may take a very long time due to the execution of software floating-point libraries. Tests and maybe even benchmark comparisons can help to find out more. So if the precision is still high enough for the application this may be an alternative for microcontrollers without floating-point unit.

The datasheet of the microcontroller also gives hints about execution times of certain instructions, for example basic arithmetic operations. For the XC2000 for instance, the division takes 21 clock cycles in order to finish (Infineon Technologies AG, 2012). On the opposite, an addition or multiplication does only take one clock cycle. Compared to this, a division is rather slow, and therefore it should be replaced by multiplications with fitted parameters and factors. Due to the lack of a floating-point unit the transformation to fixed-point code with only a few divisions has a high optimization potential for the XC2000 family. Despite this optimization is rather complex, the code generation in the example case is able to convert multiplications and divisions by powers of two into fast shift operations. The user can largely benefit from this if focussing on providing suitable factors for calculation.

Hand-optimized Code

Additional advantages in execution speed are given by C functions embedded in the CASE tool. Hand-optimized Code may lead to shorter runtimes especially in encapsulated blocks without having to deal with less flexibility. Simulink offers C functions for that kind of optimization. The use of embedded code is also suitable for the implementation of alternative algorithms. In order to ensure a certain speedup and also the correct execution of the code tests with these hand-written code segments should be performed on the target hardware prior to implementing them in the CASE tool.

Compiler Optimizations

There are lots of optimization potentials referring to the IDE. Chances of speeding up code can primarily be found in compiler and linker settings. That is why there is also much research work done, and some tools exist that enable the user to even examine worst case runtimes of their programs (Schwarzer, 2007). Many tools offer customized options for enabling or disabling certain optimization options. For runtime-optimized code the tradeoff between code size and speed should be consequently set to speed, although the resulting code may be larger. Additionally, the optimization level should be set to high, but not every single optimization offered by the compiler (e.g., function inlining, interprocedural register optimizations, loop transformations, etc.) may in fact lead to faster code. By comparing different settings and customizing sub-selections of the optimization options a best case can be found for the current project. A good starting point is a common predefined set such as "-O2" or "-O3". That means that different compiler optimization settings ("O") are combined when compiling the program. This is done with a certain level ("2" or "3", where "3" is the highest level supported). So "-O2" or "-O3" are the resulting parameters.

Influence of Memories

When having optimized compiler settings a look at the linker settings is worth a try, too. Although microcontrollers often place code in non-volatile Flash-ROM, sometimes there is a chance of placing code in much faster RAM at runtime. After copying the code from flash ROM to RAM at startup the code execution can be massively accelerated. Because unlike most flash ROMs RAMs do not need waitstates when being addressed by a microcontroller, they do not slow down code execution of the processor. Memory access times for comparison can be found in technical datasheets or even in the IDE settings of the project, e.g., the number of waitstates for different memories.

Parallelization

If processes have to run in parallel on the microcontroller an optimization is possible, too. But unlike desktop processors, real multiprocessing and the use of threads is not widespread on microcontrollers. Here especially peripherals can execute their tasks while the CPU is calculating something different. As an example, analogue-digital converters (ADCs) or communication interfaces (e.g., the Human Machine Interface, HMI) are able to perform tasks independently from the CPU. A modified program flow can turn this advantage into shorter execution times of the whole program by executing peripheral tasks in parallel. The CPU is not stalled until these tasks are finished.

Hardware Optimizations

Last but not least, it depends on the project if changes to the hardware are possible and useful. Increasing clock speeds should only be performed within the limits set by the hardware manufacturer. Replacing hardware often means huge efforts which have to be justified by the benefits of that exchange. There has to be awareness of additional costs, time and expenses for personnel. Occasionally there are higher clocked derivatives of one hardware family available which are even pin-compatible. In the example case the previously used XC2787 microcontroller can easily be replaced by the XC2289. With minimal effort a clock speed of 128 MHz instead of only 100 MHz is available, accelerating code execution by 28 %. The peripherals and pin connections remain nearly the same.

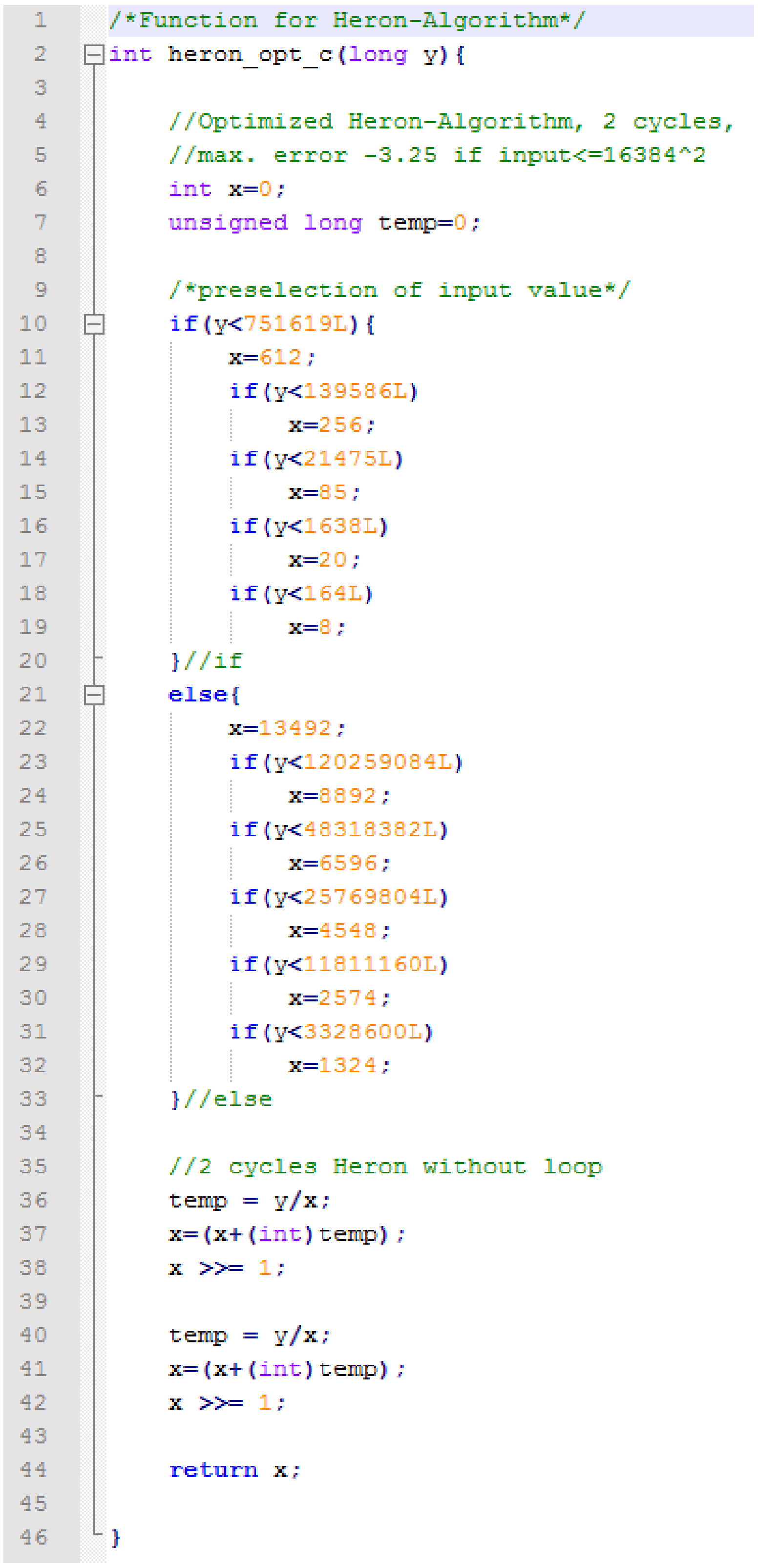
Implementation of Optimizations

The given example of the chainless bike offers initial optimizations by analysing the used Simulink model. All blocks not necessary for the control by the microcontroller are deleted. Furthermore, tests with fixed-point calculations have proven that there is no essential loss in precision when using fixed-point calculations instead of floating-point calculations. The range of values is completely used so that the loss in precision is minimized. This is achieved by adjusted factors. The factors are also matched to use fast shift operations where possible. At the same time many previously required saturations are now obsolete because the new factors prevent overflows and underflows. Using this set of optimizations it is already possible to test the control with a sampling frequency of 4 kHz. This shows the high potential of these first steps.

In addition, numerously used blocks which are built quite equally are substituted by hand-optimized C functions. This especially concerns filter and square root calculations. The conventionally used square root function "double sqrt(double)" from math.h library for example can be replaced by a highly optimized fixed-point Heron algorithm which can be executed much faster and delivers a convenient accuracy. The commonly used Heron algorithm follows this equation:

$$x_{n+1} = \frac{\left(x_n + \frac{a}{x_n} \right)}{2} \qquad (1)$$

Equation (1) shows that the algorithm contains two divisions, an addition and a multiplication. Especially the division by two can be transformed to an arithmetic right shift by one. Remembering the 21 cycles for divisions on the XC2000 this first division can now be executed in only one cycle. Furthermore, the algorithm can be optimized by setting a start value for x_0 of the iteration. Moreover, some kind of loop unrolling is possible, too. The resulting code is given in Figure 3.



```c
/*Function for Heron-Algorithm*/
int heron_opt_c(long y){

    //Optimized Heron-Algorithm, 2 cycles,
    //max. error -3.25 if input<=16384^2
    int x=0;
    unsigned long temp=0;

    /*preselection of input value*/
    if(y<751619L){
        x=612;
        if(y<139586L)
            x=256;
        if(y<21475L)
            x=85;
        if(y<1638L)
            x=20;
        if(y<164L)
            x=8;
    }//if
    else{
        x=13492;
        if(y<120259084L)
            x=8892;
        if(y<48318382L)
            x=6596;
        if(y<25769804L)
            x=4548;
        if(y<11811160L)
            x=2574;
        if(y<3328600L)
            x=1324;
    }//else

    //2 cycles Heron without loop
    temp = y/x;
    x=(x+(int)temp);
    x >>= 1;

    temp = y/x;
    x=(x+(int)temp);
    x >>= 1;

    return x;

}
```

Figure 3: Adapted Heron Algorithm

As shown in Figure 3 the adapted Heron Algorithm makes use of a preselection of the input values. By doing that, the algorithm can calculate results that are precise enough for the application in only two calculation cycles (lines 36 to 42, no loop for iteration). The start value is therefore preselected to meet the conditions of the short calculation. Standard implementations need much more cycles to calculate valid results. The required data types are limited to long and integer formats (32 Bit and 16 Bit in case of the XC2787 and XC2289). Therefore, Figure 4 shows an accuracy comparison of different Heron square root calculations.

Data Format for Algorithm	**Max. Deviation from Long Double (80 Bit)**
Double (64 Bit, sqrt())	0.000
Float (32 Bit, sqrt())	0.001
Long & Int (32 Bit and 16 Bit, fixed point heron())	3.235

Figure 4: Max. deviation vs. data format comparing different data formats to 80 Bit reference calculation

Although the calculation precision with long and integer data format is not as high as when calculating with float or double values, a maximum of 3.235 is the highest deviation occurring over the whole long value input space. The accuracy is still high enough to perform the control algorithm precisely. The overall time savings are quite high because the optimizations apply to many blocks in Simulink. This fact is illustrated in the following figure. The listed times are measured runtimes for the XC2787 operating at 100MHz and executing the program from Flash-ROM with the -O3-compiler option turned on (Figure 5).

Data Format for Algorithm	**Time for Calculation**
Long Double (80 Bit, sqrt())	47,20 µs
Double (64 Bit, sqrt())	43,60 µs
Float (32 Bit, sqrt())	36,00 µs
Long and Int (32/16 Bit fixed point heron())	4,62 µs

Figure 5: Comparison of precisions and runtimes

The costs for the implementation are low so the hand-optimized Heron-algorithm is very effective when implemented.

Various compiler optimizations also contribute to lower execution times. The settings are chosen with regard to the predefined "-O3" setting, with some minor custom flags being set. Nevertheless, the explicit use of the so-called multiply-and-accumulate unit (MAC, a special hardware feature of microcontrollers (Infineon Technologies AG, 2006, 2007)) does not provide any benefits for the runtime in this example. But several linker optimizations pay out well. By using special sections parts of the code can be executed from RAM instead of flash ROM, and so the code can be executed faster. All things considered the linker optimizations made the code execute faster by about 30%.

Further optimizations are taken – as described – by using peripherals of the microcontroller in parallel to the normal code execution (see Figure 6). Despite of the single core architecture of the XC2000 family the conversion of analogue values by the ADC and the communication can be performed that way.

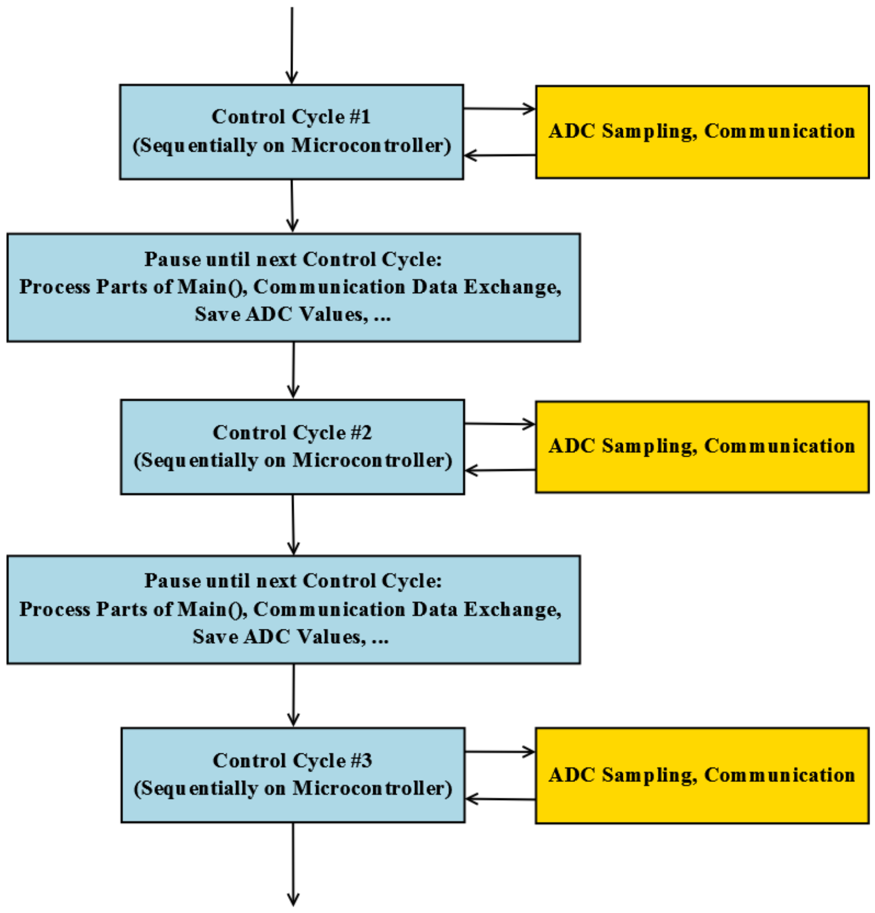

Control Cycle #1
(Sequentially on Microcontroller)
ADC Sampling, Communication
Pause until next Control Cycle:
Process Parts of Main(), Communication Data Exchange,
Save ADC Values, ...
Control Cycle #2
(Sequentially on Microcontroller)
ADC Sampling, Communication
Pause until next Control Cycle:
Process Parts of Main(), Communication Data Exchange,
Save ADC Values, ...
Control Cycle #3
(Sequentially on Microcontroller)
ADC Sampling, Communication

Figure 6: Parallel Execution of Peripheral Tasks

The time available for data exchange for the human machine communication is longer than the spare time between two control cycles of the system. That is why the communication is executed cycle-sequentially. That means a reference clock is derived from the control clock, and in each pause between the end of the current and the beginning of the next control cycle a part of the communication data is exchanged with the peripherals. After a certain period of time the communication data is transferred and new communication data can be processed. On the one hand this methodology saves valuable processing time of the CPU, on the other hand an exact execution timing of all software modules is guaranteed.

Altogether the optimizations enabled a sampling time of 101 µs. The times for performing the data exchange with communication peripherals and saving converted ADC values do not add to this time because they are processed between the control cycles. What remains is the substitution of the XC2787 by the XC2289 derivative with 28 % higher clock speed. As expected the sampling time is now about 73 µs which means the sampling time perfectly scales with the clock speed. Also there is enough time left to do the communication data exchange and save the ADC values. Figure 7 summarizes some selected optimizations and their effect on the runtime. Listing all improvements achieved by the mentioned optimizations would go far beyond the scope of the paper.

Optimization	Time saved
Model Optimization, Fixed Point Artithmetic	several ms
Parallelization of Peripheral Actions	28 µs
Adapted Heron Algo-rithm (multiple issue)	17 µs
Optimized Filter Func-tions (multiple issue)	ca. 56 µs
XC2289 vs. XC2787 (increased clock speed)	28 % of runtime
Code Execution RAM vs. Flash-ROM	ca. 30 % of runtime

Figure 7: Runtime optimization summary

The model optimization and the change of the data format to fixed point had the highest impact on runtime. Without that any other optimization would not have made sense. Several software optimizations (Heron and filter functions) could be implemented quite easily and payed off well due to the widespread use of them. The increased clock speed as well as the code execution from RAM speeded up the whole program and were not limited to certain algorithms.

Now the sampling frequency can be raised to 10 kHz. This massively improves the quality of the control and lets the bike run smoothly and silently.

Conclusions and Future Research

The example project shows how important the use of CASE tools is for engineers. In this context specific optimization is significant although flexibility and test cases of the CASE tool are important, too. Starting with the software model the optimization approaches show how runtime-optimized code can be produced. Therefore, widespread optimization steps of nearly every participating component are necessary. The efforts are not negligible. That is why the goal for further research should be the automation of the optimization processes. Because of project specific constraints this appears to be a huge challenge. A kind of plugin to the existing toolchain seems conceivable that takes the optimization constraints from the user and implements them directly into the generated code.

References

dSPACE GmbH. (2003a). *dSPACE: Control Desk. Experiment Guide*. dSPACE GmbH, Paderborn, Germany.

dSPACE GmbH. (2003b). *dSPACE: DS1103 PPC Controller Board. Installation and Configuration.* dSPACE GmbH, Paderborn, Germany.

Infineon Technologies AG. (2006). *Infineon Technologies: The Insider Guide to Planning XC166 Family Designs.* Infineon Technologies AG, Munich.

Infineon Technologies AG. (2007). *Infineon Technologies: DSP Optimization Guide for XC2000, XE166 and XC166 Microcontroller Families with MAC Unit.* Application Note 16113. Infineon Technologies AG, Munich.

Infineon Technologies AG. (2012). *Infineon Technologies: Data Sheet XC2288I, 2289I* (Edition 2012-06, p. 7-8). Infineon Technologies AG, Munich.

The Mathworks. (1997). *The Mathworks: Using Matlab.* [Software Manual].

Schwarzer, M. (2007). *Untersuchung des Einflusses von Compiler-Optimierungen auf die maximale Programmlaufzeit*. Diploma Thesis, University of Dortmund, Department of Computer Science.

Biographies

Dipl.-Ing. (FH) **Thomas Stolze** has been a research associate and lecturer in the Department of Automation and Computer Science, at Harz University, Wernigerode, since 2008. His studies include Technical Informatics, Harz University Wernigerode (Dipl.-Ing. FH) and Doctorate at Technical University of Ilmenau. His research interests include Benchmark development and system comparison, Microcontroller-, DSP- and Microprocessor technologies and applications, system evaluation.

Prof. Dr.-Ing. **Klaus-Dietrich Kramer** has been Professor for Microprocessor Systems in the Department of Automation and Computer Science at Harz University, Wernigerode, since 1998. He is also an application Engineer in an engineering institute and lecturer at the Ingenieurschule Eisleben, Since 2004 he has been president of the Institute of Automation and Informatics (IAI) in Wernigerode. His studies include Information Technology, Technical University of Dresden (Dipl.-Ing.), Promotion (Dr.-Ing.) at Technical University of Ilmenau. His research interests are Microcontroller applications, benchmarks for Microcontrollers, Microprocessors and Digital Signal Processors, CI-Applications (Fuzzy- Control and Low-Cost-MC, Real Time CI-Systems, etc.), and Automotive applications.