# Profiling of Modelica Real-Time Models

Christian Schulze[1] Michaela Huhn[1] Martin Schüler[2]

[1]Technische Universität Clausthal, Institut für Informatik, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Deutschland
{Christian.Schulze | Michaela.Huhn}@tu-clausthal.de
[2]TLK-Thermo GmbH, Hans-Sommer-Str. 5, 38106 Braunschweig, Deutschland
M.Schueler@tlk-thermo.de

## Abstract

Modeling and simulation of physical systems have become a substantial part in the development of mechatronic systems. A number of usage scenarios for such models like Rapid Control Prototyping and Hardware-in-the-Loop testing require simulation in real-time. To enable model execution on a hard real-time target, a number of adaptations are usually performed on the model and the solver. However, a profiling facility is needed to direct the developer to performance bottlenecks.

We present the concepts and a prototypical implementation of a profiler for the specific analysis of Modelica models running on Scale-RT, a Linux-based real-time kernel. The profiler measures the number of calls and execution times of simulation specific functions calls. Interpreting these results, the developer can directly deduce which components of a simulation model are most promising for optimization. Profiling results and their impact on model optimization are discussed on two case studies from the area of thermodynamic automotive systems.

***Keywords*** Real-Time, Modelica, Profiling, Optimization, SimulationX, Scale-RT

## 1. Introduction

The modeling and simulation language Modelica is widely accepted in transport industries, in particular in the automotive area. Modelica is employed for modeling the physics of the controlled system in the software development process of electronic control components. Whereas so far simulation aimed for conceptual validation in the early concept phase, nowadays we find an increasing need for real-time simulation or even real-time execution of models on micro-controllers.

Prominent usages of real-time simulation are Rapid Control Prototyping (RCP) [7] and Hardware-in-the-Loop (HiL). These are techniques for the concept and development phases: The overall system is modeled as a combination of the controlled part and a model of the controller - often in a unified modeling and simulation environment. Combined simulation facilitates validation not only of the concepts, but - in a stepwise refinement process - also of the detailed functional and timing behavior of the controller under design, provided detailed physical models and sufficient computing resources are available. For this purpose, the major requirement is that simulation runs as fast as the real system. Several real-time platforms are available to support RCP or HiL, like the open-source Linux-based Scale-RT [9] running on standard PCs, or specific hardware solutions e.g. dSPACE systems.

Another usage of real-time simulation is to execute the model of the controlled system as part of the control: The idea of Model Predictive Control (MPC) is to predict the short term behavior of the physical system by feeding the sensored data from the system into the model and simulate its reaction on possible inputs from the controller, thereby optimizing the controller strategy. In on-board diagnostics, the results from a model running in parallel on the controller are compared to the measurements of the real system to deduce abnormal behavior that is a sign of failures. For these usages, the simulation model has to be executed on the same target as the control, i.e. a micro-controller with restricted resources in many cases. Consequently, being part of the control component imposes hard real-time constraints on model execution.

The usages we mentioned are in the context of *hard* real-time systems (HRT), i.e. systems for which the timely response has to be verified for *all* possible executions of a system component. In contrast to hard real-time, a *soft* real-time system is only required to perform its tasks according to a desired time schedule on the average [3]. As a consequence of the stringent needs for verification, the component behavior of HRT systems has to be analyzed in detail with respect to the timing constraints.

In order to guarantee predictable execution times, simulations on real-time targets typically use a fixed-step solver to solve the DAE-System (Differential Algebraic Equation). Moreover, such models require highly efficient modeling to not exceed the given step size. But even then source code that is automatically generated from Modelica models may violate the timing constraints.

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

During the solving process the solver generates events for every zero crossing of a zero function. The solver examines the DAE-system in an interval close about these events, so events cause additional work load and will increase the model runtime.

So, timing problems of the simulation model may arise from various causes like events or the internal complexity of the model. They become evident when runtime exceeds the solver step size and hence the HRT-system usually will abort the execution (although there are ways to construct solvers that are able to ignore such overruns (see [14]).

In case of a timing violation the developer of the simulation model has to improve the model's efficiency by reducing the model's complexity, by setting better start values, or other measures. But so far these steps are purely based on the developer's experience. Real-time profiling may help to give him or her a better understanding of the underlying DAE-System since Kernighan and Pike note "Measurement is a crucial component of performance improvement since reasoning and intuition are fallible guides and must be supplemented with tools like timing commands and profilers." [8]. Based on profiling results the developer is able to identify the components causing the main work load and decide whether a submodel has to be enhanced or an algebraic loop has to be broken.

There are several real-time profiling tools available in particular for Linux-based systems [12, 2], but these are general purpose profiles and not specifically well suited to analyze code for real-time targets that was automatically generated from tools like Dymola or SimulationX. First of all, the actual profiling shall be performed on the real-time target itself to get direct information about the runtime on a specific host. Profiling an execution on a standard PC under Windows and scaling the results for a specific target as it can be done in other domains will not give valid approximations here, because most approaches and tools employ another (fixed-step) solver for execution on real-time targets whereas variable step solvers are used under Windows which differ significantly with respect to their timing characteristics. In addition, the real-time target may impose further restrictions on the profiling, e.g. the real-time Linux Scale-RT 4.1.2 compiles and runs the model as a Kernel-Module.

The described usage scenario in the development of simulation models requires a precise measurement of function execution times and function call counting as well as measurement of certain code sections representing algebraic loops lead to the development of a new profiling tool that can be used on such a real-time operating system.

Within the source code of a model calls to external libraries may occur, e.g. fluid property libraries. For modeling of thermodynamic systems most of the work load is generated by those function calls. Therefore it is necessary to examine them closer.

In general, an algebraic loop results from connecting the output of a submodel to the input of that particular model. Due to this cyclic dependency relation, models containing algebraic loops have to be solved iteratively. Algebraic loops cause serious problems in simulation tools based on a simple input-output-block structure like Matlab Simulink. In equation based modeling the loops are traced back to the underlying equations and may be solved analytically but still many of them have to be solved numerically[13].

The profiling method introduced in this paper will measure the execution time of each function call, to count function calls separately in each relevant section and to measure the time needed to solve algebraic loops, so called "(non-)linear blocks". Especially for profiling of real-time models the overhead of the profiling method on the model runtime shall be kept small. This is achieved by implementing the producer-consumer-pattern as described in Section 4.

Basically this concept can be applied to each target operating system and simulation environment (e.g. Dymola). Even an online evaluation of the profiling results during execution of the model could be implemented. Until now we implemented this concept for models exported from SimulationX to Scale-RT. The instrumentation for the case studies has been done manually, but automatisation will be finalized soon.

## 2. Profiling on Real-Time Targets

Tracing and profiling are two related techniques that aid the developer to understand the behavior of a program. Tracing gives a detailed view on which function is called, who is the callee, how long does the execution take and also a call counting may take place. Profiling instead gives a statistical evaluation of average execution times and frequencies of the function calls or the profiled sections [12].

The tracing results on a particular program execution can be displayed as call-chains in a call-graph. A call-graph demonstrates the possibly complex call structures annotated with the execution times of the callee. Call-graphs are especially helpful to understand the communication of threads within multi-threaded applications. A Profiling tool generates simpler, statistical results without any structure or evaluation of the call context. However, as the execution of a simulation model follows a fixed elementary plan as described in Section 3, profiling is sufficient for our purposes. Profiling and tracing generally involve three phases:

- instrumentation or modification of the application to perform the measurement
- actual measurement during execution of the application
- analysis of results

Instrumentation adds instructions to the application for measuring execution times, updating counter variables and measuring the consumption of resources. Instrumentation cannot only be performed at source code level but also during compilation, linking or even at the target code level. Where the instrumentation takes place depends on the profiling tool. However, in any case the measured data need to be traced back to the source code level for interpretation.

The instrumentation will obviously increase the execution time of the application, because additional steps will be taken to measure and store the profiling or tracing data.

In general, the overhead caused by instrumentation shall be reduced to a minimum.

The two main approaches to profiling are based on sampled process timing and measured process timing. In profiling based on sampling, a hardware interval timer periodically interrupts the execution of the profiled application. During interruption the profiling tool examines which parts of the program have been executed since last interruption. Profiling tools like prof [5] and gprof [4] are based on sampling and commonly employed.

In profiling based on measured process timing the instrumentation procedures are called at function entry and exit. When entering or leaving a function a time stamp is recorded additionally to counters and timers. Profiling tools like TAU [11] are based on this approach and we will follow it, too.

## 3. Profiling of Modelica Models

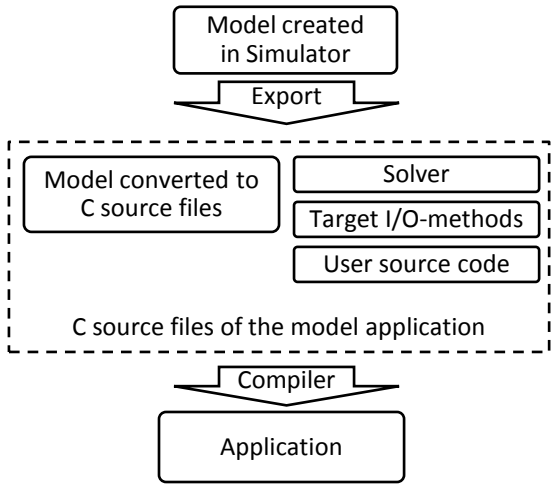


**Figure 1.** Export of models from a simulator to target

Simulation tools like SimulationX, Dymola or Matlab/Simulink Realtime Workshop share the basic process of exporting models which is depicted in Figure 1. The simulator transforms the model to a DAE-System and after that into C code written in a file. Several other sources are added to the transformed model including a mathematical solver and target specific I/O-methods, but those additional files are always the same. Furthermore, user code can be integrated manually at this point introducing user methods or user libraries. This set of files builds the source code for an application that calculates the desired results.

The C code originating from the models that were transformed by the simulator have a common simple structure. In the moment, this is specific to the framework used for simulation, but with the new Functional Mock-up Interface (FMI) [1] this is standardized. The FMI defines the external interface between the C-Code or even the target code generated from models and the solvers. By using the FMI interface, models can be connected to any solver that realizes the solver's side of the interface; and vice versa a solver may be attached to any model that communicates via FMI. Thereby models can be executed in "foreign" simulation frameworks. When connecting the model code to a solver which is part of the native simulator another internal and more efficient interface may be used. However, in our current approach the proprietary format of the model C code provided by Simulation X is taken as input for the profiling.

To execute the model the following steps are taken:

- export of the model
- compilation of the model application
- transfer to the real-time target
- execution

The work flow for Scale-RT 4.1.2 as the target system is as follows: SimulationX compiles the source of the model in the Cygwin environment, which has to be installed under Windows. This environment provides all libraries and includes needed by this version of Scale-RT. The resulting file is a tgz-file containing the compiled model and additional settings.

In order to be executed on the Scale-RT, the model application is sent to the target system, e.g. by using the Scale-RT Suite, which is part of the Scale-RT Environment as well as Cygwin. Subsequently the model application can be executed from the Scale-RT Suite. SimulationX is able to send the model application to the target system and execute it, too, but the results cannot be observed from there.

In general, the model is separated in an initialization and the simulation problem. Both of them consist of a number of integration steps as well as a set of explicit calculations of the outputs. A global fixed-step solver is used on real-time targets for solving. In case of an overrun the execution stops; so it is considered as a hard real-time simulation guaranteeing the delivery of results within a certain time.

Each step of the global solver consists of one method called several times representing the integration step and one method called only once outputting the simulation results through defined I/O-methods. Within those two methods every calculation including function calls to user libraries occur. Within the integration steps (non-)linear equations (algebraic loops) that could not be solved analytically are evaluated numerically using a local solver. So the structure of the source code for both, the initialization and the simulation problem, looks as follows:

- Global solver step
  - $n_I \cdot$ integration steps
    - $e_I \cdot$ external function calls
    - $c_I \cdot$ additional calculations
    - $a_I \cdot$ (non-)linear blocks
      - $e_{aI} \cdot$ external function calls
      - $c_{aI} \cdot$ additional calculations
  - $1 \cdot$ output of variables
    - $e_O \cdot$ external function calls
    - $a_O \cdot$ additional calculations

Because of the flat structure of the automatically generated source code the call-chain is not needed to understand and analyze performance bottlenecks in a model in many cases. We consider that a flat profiling for each relevant section as the best choice. A flat profiling should be performed on each (non-)linear block within the integration step as well as the integration step and the output of variables itself. This gives the clearest view on the work load caused by every single section.

Since hard real-time simulation shall guarantee the delivery of the results within the time limits, the maximum runtime of a model is more important than the average runtime. The profiling methods described in this paper measure and save the execution times of each simulation step separately. Then the average and variance as well as the maximum of the execution times are determined.

The aim of profiling is to direct the developer towards parts of the model that are worth optimizing. But the developer is in charge of optimizing the model manually. However, after optimization it has not only to be verified that the model application is running faster as before indeed, but also that the optimized model calculates its results with sufficient accuracy. So the work flow of real-time optimization is as follows:

1. check model runtime, if real-time constraints are satisfied finish optimization
2. perform profiling
3. analyze the profiling data and identify performance bottlenecks worth optimizing
4. optimize the model
5. check correctness of modification, if deviation errors are too big revert and go back to 3.
6. go back to 1.

## 4. Implementation

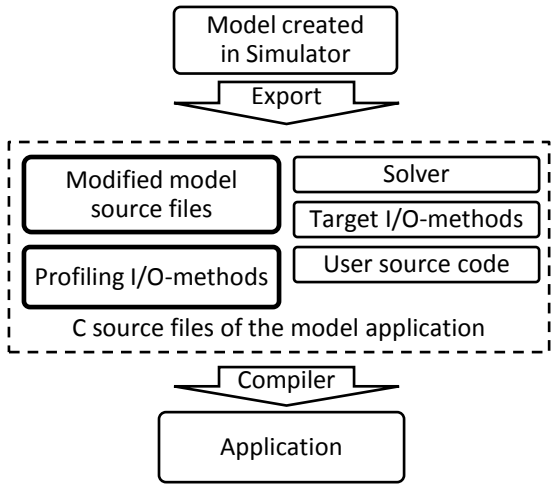


**Figure 2.** Modified export of models from a simulator to target

Comparing figure 1 and figure 2 reveals the modifications necessary for profiling. The converted model C source file and the file containing the main real-time routines have been modified to perform the profiling, access a FIFO buffer and to provide buffer memory to store the profiling data internally.

As explained, the contribution of the profiling on execution times shall be as small as possible. In our context where the model will be executed as a kernel model, an efficient solution for outputing profiling data is a major point for minimizing the overhead. Therefore we applied the consumer-producer pattern and divided the profiling into two tasks: The real-time Kernel task executing the model and a User Space task - outside the hard real-time context of the kernel - evaluating and storing the measured data. The two tasks are communicating through first-in-first-out (FIFO) buffer.

As displayed in figure 3 the source code for the model application is compiled as real-time task kernel module in Scale-RT. The goal of instrumentation of the kernel task is to log each external function call's execution time in detail. The instrumentation can easily be automated. For different analysis szenarios the instrumentation is configured to profile only the functions calls and sections of interest. For the case studies described in Section 5 insturmentation has to be done manually, but automatisation will be finished soon.

In order to store the profiling data on the hard disk without delaying the execution of the model, a consumer is created reading the FIFO buffer, interpreting the data and storing statistical data, the minima and maxima for each global solver step on the hark disk. As the execution times of different sections as well as the external function calls are measured the overhead of the global solver as well as the profiling overhead can be estimated by comparing it to the performance of the uninstrumented model.

For profiling, the model allocates memory of a fixed size for an intermediate buffer and a main buffer when it begins to execute. The main buffer is used to store the data until the non-real-time user task can process it. This buffer is implemented as a double buffer to avoid buffer overflows. As soon as the first buffer is filled the routine switches to the secondary buffer and sends the content of the first one to the FIFO buffer. The intermediate buffer is used to record all profiling data of one global solver step and is emptied in the main buffer at the end of the current step.

Since in version 4.1.2 of Scale-RT in combination with Cygwin the Kernel-Module cannot export symbols to the user address space, the main buffer cannot be accessed directly by the user task. Writing the data into the virtual file system procfs a.k.a FIFO buffer is a temporary workaround for this problem enabling the Kernel task to store data efficiently. The user task triggers the execution of a Kernel tasks method which copies the main buffer into the FIFO buffer. In future the main buffer will be accessed and read out by the user task directly, therefore these buffers will use shared memory allocation methods.

For each global solver step of the model, the profiling methods record the following information:

- execution time and frequency of an integration step

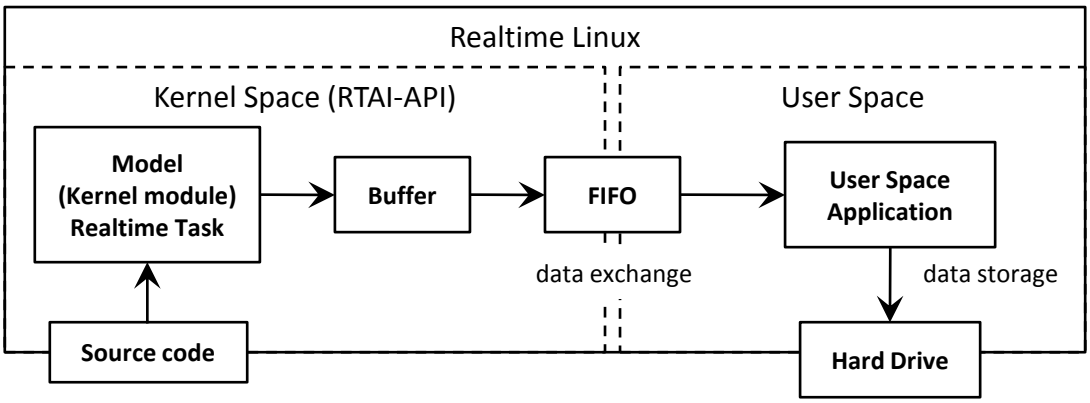


**Figure 3.** Communication between user task and model task

- execution time and frequency of each external function call within the integration step that does not reside inside a (non-)linear block
- execution time and number of loops of each (non-)linear block within the integration step
- execution time and frequency of each external function call within each (non-)linear block
- execution time of outputting variables at the end of the current step

## 5. Case Studies

### 5.1 Case 1: Moist Air inside the Cabin of a Car

#### 5.1.1 Description

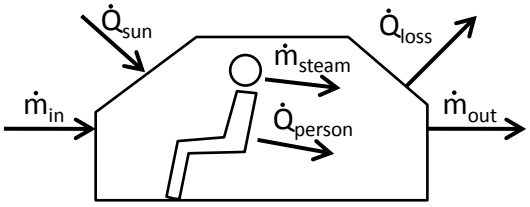


**Figure 4.** Model of the air inside a cars passenger cabin

This Modelica model describes a simple system of the air within a cabin of a car as displayed in figure 4 and it uses the TEMO-property-library.

There is a mass flow of moist air entering the system coming from the air conditioning and another mass flow of moist air leaving the system. There is an heat flow induced by the sun heating up the air inside the cabin and another heat flow out of the cabin due to heat losses. In addition to that there is a person inside the cabin who heats up the air and also increases the moisture by a given water mass flow.

The thermodynamic properties of the moist air are modelled on the basis of the ideal gas theory [15]. Condensing and simple frost formation can be described with the property equations given below. As the pressure in this case study is about $1bar$ with temperatures down to $0°C$ the error introduced by applying the ideal gas theory is very small.

$$\begin{aligned} \frac{dm}{dt}(h - pv) + & \\ (c_p - R_i) \cdot \frac{dT}{dt} \cdot m &= \dot{m}_{in} \cdot h_{in} \\ &\quad + \dot{m}_{steam} \cdot h_{steam} \\ &\quad - \dot{m_{out}} \cdot h_{out} \\ &\quad + \dot{Q}_{sun} + \dot{Q}_{person} \qquad (1) \\ \frac{dm}{dt} &= \dot{m}_{in} + \dot{m}_{steam} - \dot{m}_{out} \qquad (2) \\ \frac{dm_{steam}}{dt} &= \dot{m}_{in} \cdot \xi_{in} + \dot{m}_{steam} \\ &\quad - \dot{m}_{out} \cdot \xi_{out} \qquad (3) \end{aligned}$$

Equation (1) is the first law of thermodynamics applied to this model. The left side represents the dynamic change of energy inside the cabin, the right side embodies the heat and mass flows into and out of the cabin.

The pure mass balance is described in equation (2), the balance for the water inside the cabin is defined by equation (3). The concentration of water inside the air can be calculated using these definitions.

This model has been developed during development of the real-time TEMO-property-library, so it was used to optimize the structure and interface of the library and has been optimized several times. As a result, the number of calculations is reduced to a minimum.

#### 5.1.2 Results

Figure 5 shows the execution time of a global solver step split into the main contributors described on page 3. The four integration steps cause almost 98% of the work load. The output of the results can be neglected as it causes less than 1% of the work load. The summing up the integration steps does not equal the model runtime, as the global solver still has to evaluate the results and to perform auxiliary operations. The profiling itself increases the gap between the sum of each single contribution and the total runtime of the model, but this manipulation cannot be avoided.

The impact of profiling on the execution times can be estimated by capturing the total runtime of the whole model before and after the instrumentation. If every external function call and every (non-)linear block is profiled then the

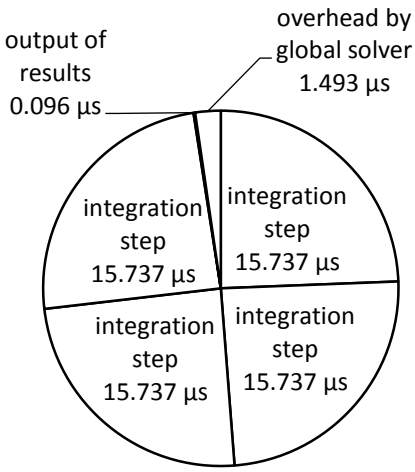


**Figure 5.** Integration steps cause main work load in the global solver steps

runtime of the model in both case studies increases by 4% at maximum.

As the external functions are called within the integration step, the execution time of the integration step inherits the profiling overhead caused by the external function calls. Therefore the gap between the sum of the executions times of integration steps and the output of variables is not as big as the increase of the whole model runtime. Because of this the overhead displayed in figure 5 can partially be assigned to the global solver.

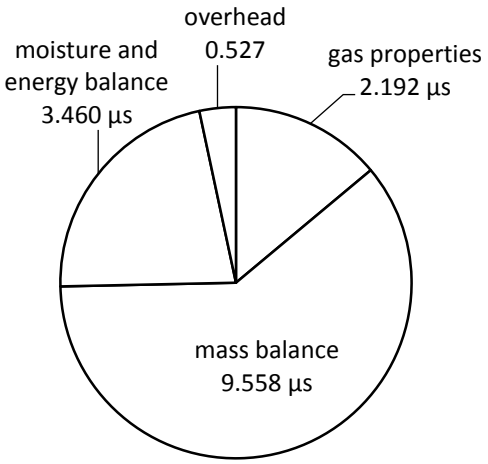


**Figure 6.** Work load of an integration step broken down to contributions

Figure 6 details the partitioning of the execution times within the global solvers integration step. There are just two algebraic loops, one representing the mass balance and another embodying the moisture and energy balance. The calls to the external gas property functions only occur outside the algebraic loops. The gap between these three summands and the execution time of one integration step is equal to the overhead. This overhead contains all calculations that are not external function calls and not algebraic loops. The rest is caused by the profiling methods.

The described model is already optimized so there is no algebraic loop or external function call causing extraordinary model runtime anymore. Figure 6 shows the balanced sharing of the given step time.

### 5.2 Case 2: Steady State Continuity

#### 5.2.1 Description

This case was built up in Modelica using the real-time TEMO-property-library with the TIL-Library by TLK-Thermo and the Institute for Thermodynamics of the Technische Universität Braunschweig [10, 6].

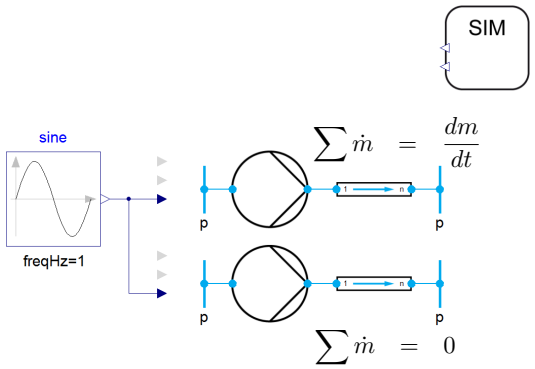


**Figure 7.** Non-linear pump and tube models with and without thermal expansion of incompressible liquid

As visualized in figure 7 the model is composed of two boundaries, a pump and a tube. The medium used in this case is incompressible water, so all fluid properties only depend on the temperature. Each property can be calculated using a Modelica function of temperature. The pressure increases at the pump is a second order function of the volume flow rate. The tube model is based on the finite volume concept and here composed of 2 cells. Within every cell there is a mass-, energy- and momentum-balance. The sine curve source sets the temperature at the inlet of the system. The temperature changes with an amplitude of $5K$ and an offset of $300K$.

In each component a parameter called "SteadyStateContinuity" is introduced by the TIL-Library. This parameter switches the mass balance of that component. In steady state the amount of mass flowing into a component equals the flow out at the same time (5). But in dynamic scenarios a mass flow is induced by a change of temperature due to the expansion of the fluid (7). The isobaric expansion coefficient $\beta$ can be used to describe the expansion of a fluid due to temperature change (4). For incompressible liquids the density is not dependent on the pressure, so the change of density can directly be related to $\beta$.

$$\beta = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p \tag{4}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} \tag{5}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} - V \cdot \rho \cdot \beta \cdot \frac{dT}{dt} \tag{6}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} + V \cdot \frac{d\rho}{dt} \tag{7}$$

If a submodel for a component uses dynamic state continuity, the mass flow is directly related to the change of temperature. The DAE-system generated from this model must take this relation into account and hence the simulator has to increase the complexity of the DAE-system.

The change of density due to the change of temperature can be neglected in most cases of dynamic simulation since this effect is not relevant to the overall results of the whole model. By activating the steady state continuity the mass balance is not fulfilled anymore and mass may appear or disappear, but the main algebraic loop is broken into several smaller ones. There is no direct connection between mass balance and energy balance anymore, so the underlying smaller algebraic loops can be solved separately. This trick reduces the size of the DAE-System in paricular for the simulations of cycles.

For comparison, two subsystems with a tube and a pump were instantiated, where one is using the steady state continuity equation while the other one is not. The profiling should expose the work load caused by computing a negligible effect of density change by temperature.

### 5.2.2 Results

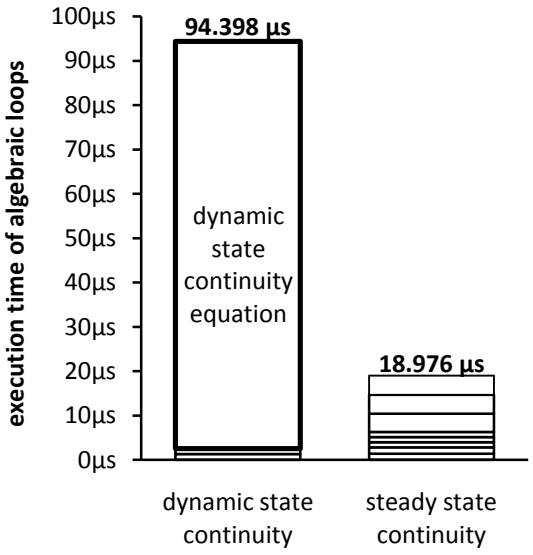


**Figure 8.** Profiling aids identifying the critical algebraic loop causing the main work load

Figure 8 visualizes the contributions of the each algebraic loop to the whole integration step separately for the steady state continuity submodel and the dynamic state continuity submodel. It allows the user to identify the critical calculations. The major work load in the submodel using dynamic state continuity equation is caused by that particular continuity equation. This algebraic loop generated from that equation has to be broken to reduce the execution time of this sub model. The simplification using steady state continuity is a method to break this loop into several smaller loops which can be solved more quickly.

Both models are equivalent in their results but differ with respect to their performance. As the global symbolic analysis performed during export selects different state

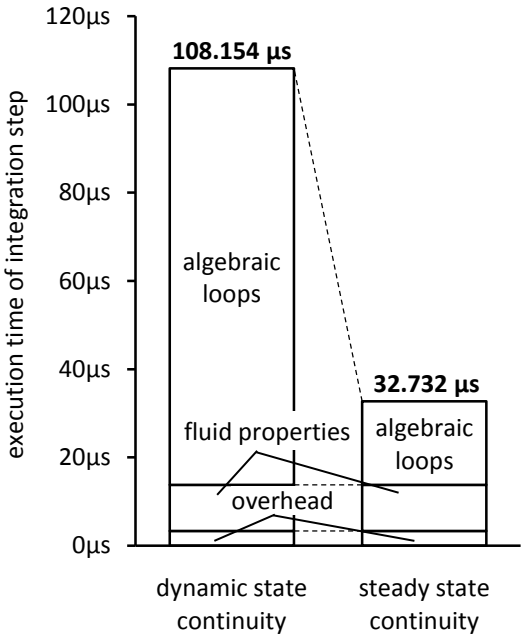


**Figure 9.** Solving times for algebraic loops in integration step of steady state continuity model are clearly faster

variables for each model, the contributions by the single algebraic loops cannot be related directly to the corresponding algebraic loops in the other model. The only way to link the algebraic loops back to the underlying equations is to trace back the involved variables.

These models were built using thermodynamic property functions which provide properties as a external function of temperature. This may cause additional algebraic loops if inverse calculation is needed, e.g. for finding the corresponding temperature to a given enthalpy. To avoid this the temperature inside the finite volumes of the tube is described as a differential state. As a result there is no algebraic loop including calls to the fluid property functions in both models.

Figure 9 relates the two models with respect to the work load caused by algebraic loops to the external function calls and the overhead of the global solver. The overhead and the amount of fluid property calculations is the same for both submodels. The contribution to the execution time of the integration step by the steady state submodel is significantly smaller.

There are other ways to break algebraic loops in a model, if the resulting relation between the variables embodies no or less important physical effects. For example a capacitor can be used to decouple the direct dependency between variables introducing a new differential state variable. Many physical models idealize a system that normally contains capacitors (e.g. the expansion of a tube due to a pressure increase) that have been neglected. Although the capacity may be very small, the effect is an uncoupling of the algebraic loops.

Figure 10 visualizes the mass flow at both sinks. The change of temperature at the inlet leads to a change of mass flow rate. In case of the dynamic state continuity equation

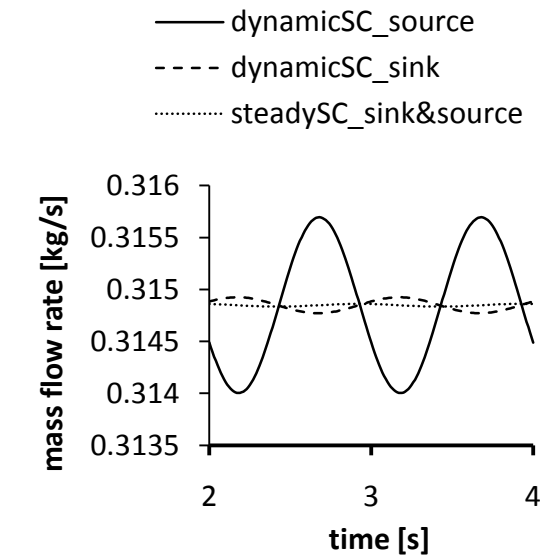


**Figure 10.** Error in mass flow due to usage of steady state continuity equation

the mass flow rate at the inlet is not equal to the outlet as a result of the expansion of the liquid. In case of the steady state continuity equation the mass flow entering all components is equal to the mass leaving the system and hence this also applies to the whole system.

The deviation between the mass flow rate entering and leaving those systems is smaller than $0.3\%$. So the simplification of using steady state continuity equation for dynamic state simulation is hardly affecting the results.

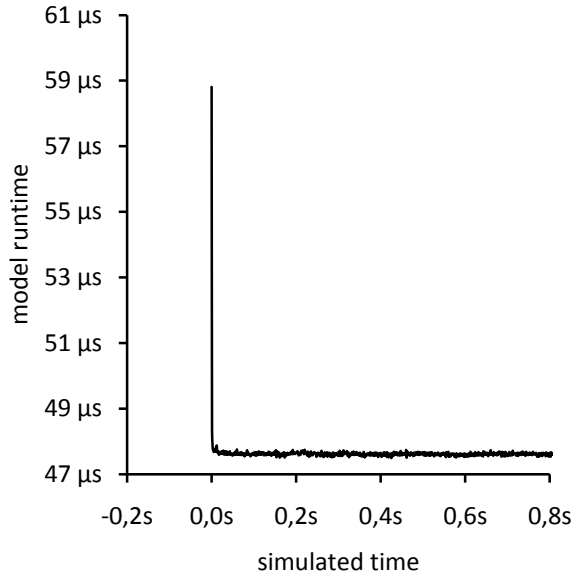


**Figure 11.** Model runtime during initialization is the bottle neck

The Figure 11 illustrates the temporal variation of the model runtime. After that first peak of $59\mu s$ during initialization of the model the runtime resides at a constantly lower level of $48\mu s$. There are no bigger changes or events inside the model after the initialization process. This case study was performed on a common Desktop PC with a Intel Pentium 4/ 540 CPU at 3.2 GHz without any realtime I/O-Interfaces.

## 6. Conclusion

This paper presents a brief description how profiling on source code that was automatically generated from Modelica tools like SimulationX can be performed under the target real-time operating system. Profiling can be a powerful tool aiding the user to understand the work load contributions by the internal algebraic loops. For optimization of Modelica models in general profiling should be introduced as a standard tool.

### Acknowledgments

This work was funded by the Federal Ministry of Education and Research (BMBF), Germany, in the project TEMO (grant 01lS08013C).

We are thankful to Adina Aniculăesei for implementation support.

## References

[1] MODELISAR (ITEA 2 07006). Functional mock-up interface for model exchange, January 26 2010.

[2] Tim Bird. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*, 2009.

[3] L. Dozio and P. Mantegazza. Linux real time application interface (rtai) in low cost high performance motion control. *Motion Control 2003*, 2003. Milano, Italy.

[4] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *Proceedings of SIGPLAN Ś82 Symposium on Compiler Construction*, volume 17, number 6, pages 120–126, June 1982. SIGPLAN Notices.

[5] S. Graham, P. Kessler, and M. McKusick. An execution profiler for modular programs. In *Software - Practice and Experience*, volume 13, pages 671–685, 1991.

[6] M. Gräber, K. Kosowski, C. Richter, and W. Tegethoff. Modeling of heat pumps with an object-oriented model library for thermodynamic systems. In *6th Vienna International Conference on Mathematical Modelling*, Vienna, 2009. ISBN 978-3-901608-35-3.

[7] K. Hoffmann, F. Heßeler, and D. Abel. Rapid control prototyping with dymola and matlab for a model predictive control for the air path of a boosted diesel engine. In *E-COSM - Rencontres Scientifiques de l'IFP*, pages 25–33. Institut Francais du Petrole, 2006.

[8] Brian Kernighan and Rob Pike. Finding performance improvements: Excerpt from the practice of programming. *IEEE Software*, 16(2):61–65, 1999.

[9] Cosateq GmbH & Co. KG. Scale-RT, 2010.

[10] Christoph C. Richter. *Proposal of New Object-Oriented Equation-Based Model Libraries for Thermodynamic Systems*. PhD thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2008.

[11] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using c++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145. ACM, August 1998.

[12] Sameer Shende. Profiling and tracing in linux. In *Proceedings of Extreme Linux Workshop*, 1999.

[13] Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattsson. Evolution of continuous-time modeling and simulation. In *The 12th European Simulation Multiconference*, Manchester, UK, June 16 - 19 1998.

[14] Inc. The Mathworks. Execution and real-time implementation of a temporary overrun scheduler, 2006.

[15] VDI. Thermodynamische Stoffwerte von feuchter Luft und Verbrennungsgasen. *VDI-Handbuch Energietechnik*, 2000. VDI Richtlinie 4670.