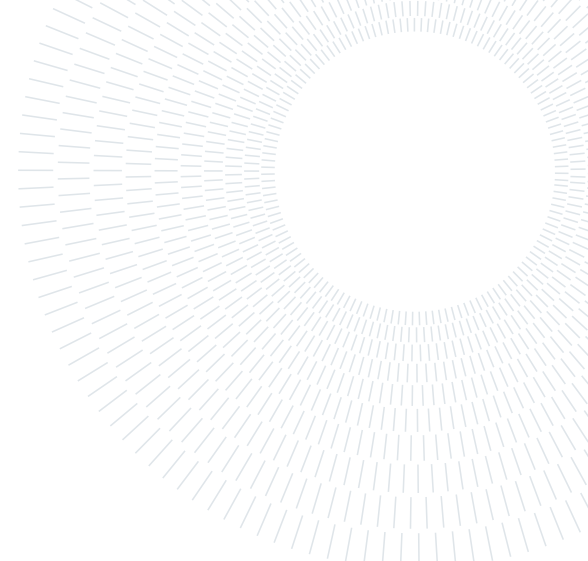

POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Compilation and Optimization of Large Scale Modelica DAE Models

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Nicola Camillucci

Advisor: Prof. Giovanni Agosta

Co-advisors: Francesco Casella, Daniele Cattaneo, Stefano Cherubin, Alberto Leva, Michele Scuttari, Federico Terraneo

Academic Year: 2020-2021

---

1. Introduction

Large scale physical phenomena can be modeled by means of differential and algebraic equations systems. Many engineering fields take advantage of modeling to facilitate the prototyping, verification and maintenance of any system, because it allows to know ahead of time the evolution of the behavior of these systems.
Due to these needs, a declarative modeling language called Modelica [3] was devised to describe a physical model with a high-level structure and then use this description to produce an accurate simulation. Currently, the Modelica compilers available on the market are not capable of exploiting the design features of modern computer architectures. This is caused by the inability of such compilers to take advantage of object-oriented structures that compose the model.
For this reason, a new compiler has been designed in recent years to overcome these limitations. This compiler, called MARCO, has the main objective of preserving as much as possible any vectorial data structures used in describing the model [1, 4]. Doing so, it is possible to fully exploit the advantages of data-locality-based features of modern computers such as caches.
MARCO was initially only capable of simulating models through the use of the Forward Euler method. This poses great restrictions on the type of models that can be simulated using this compiler. This document describes how this newly born compiler has been enhanced with an external DAE solver, called IDA, while also maintaining the original objective of preserving the data structures of the physical model. This way, MARCO will be able to create an efficient simulation without posing any restriction on the input models.

2. Background

Our work involves concepts of differential analysis and software engineering, and in particular the design and implementation of compilers. The relevant aspects of these fields are refreshed in this section.

2.1. ODE and DAE Models

In the most generic form, a Differential Algebraic system of Equations can be written as:

$$\boldsymbol{F}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) = 0 \tag{1}$$

where the derivative vector $\dot{\boldsymbol{x}}$ and algebraic variables vector $\boldsymbol{v}$ represent the unknowns of the system, while both the state vector $\dot{\boldsymbol{x}}$ and the input vector $\boldsymbol{u}$ are always known at any instant

of time.
In some cases, a DAE system can be translated into a simpler form called Ordinary Differential Equation (ODE) system:

$$\begin{cases} \dot{\boldsymbol{x}}(t) &= \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \\ \boldsymbol{v}(t) &= \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \\ \boldsymbol{x}(t_0) &= \boldsymbol{x_0} \end{cases} \tag{2}$$

By adding the third equation, which represents the initial conditions of the system, we obtained an Initial Value Problem that can be simulated for a given interval of time. The equations inside an ODE model can always be explicitated and reordered so that their Incidence Matrix (IM) becomes a Lower Triangular (LT) matrix. Doing so, all the equations can be then translated into a series of assignments that solve the model.
To algorithmically compute the evolution of an ODE system, we also need to know how the state variables evolve through time. The simplest way to do this is the first-order explicit Forward Euler formula. Unfortunately, this method is not suitable to solve every DAE model. In particular, the presence of algebraic loops or implicit subsystems pose great difficulties in transforming the model into an ODE form. This is because the IM of the system becomes a Block Lower Triangular (BLT) matrix, and therefore it cannot be ordered and scheduled.
If we want to solve a DAE model without first translating it in ODE form, we need an implicit algorithm that can handle differential equation, algebraic implicit nonlinear equation and algebraic loops altogether. This integration mechanisms are called DAE solvers. For the purpose of this document, the IDA solver contained in the SUNDIALS suite [2] will be used. It implements a variant of the DASSL algorithm, which uses a variable order and variable step size BDF formula.

2.2. Modelica Language

Modelica [3] is a declarative, object-oriented, multi-domain modeling language, developed for component-oriented modeling of complex systems. It allows users to model physical systems using a set of variables, differential and algebraic equations. Differently from imperative languages, the equal sign = does not represent an assignment operation nor it states causality among variables. It is just the declaration of an equation.

Listing 1 Heat transfer in a 1D wire

```
model ThermalWire

  // Parameter declarations [...]

  parameter Integer nx = 10;
  Real[nx] T(each start = Tlow);
  Real[nx+1] Tb;
  Real[nx+1] Qb;

equation
  for x in 1:nx loop
    c * der(T[x]) = Qb[x] - Qb[x+1];
    Qb[x]    = 2*g*(Tb[x] - T[x]);
    Qb[x+1] = 2*g*(T[x]   - Tb[x+1]);
  end for;
  Tb[1] = Thigh;
  Tb[nx+1] = Tlow;
end ThermalWire;
```

Modelica also allows users to declare parametric multidimensional arrays, for example to discretize the length of a wire as shown in listing 1. This is effectively equivalent to a declaration of several scalar variables at once. We can also perform vector operation using such arrays, and iterate through the arrays with the *for* statement. Similarly to the equal sign, for-loops in Modelica do not represent control flow inside the simulation, they are simply a way to declare multiple equations with fewer statements and in a more readable way.

2.3. Modelica Compilers

The two most used Modelica compilers are currently the OpenModelica Compiler (OMC) and Dymola. Both of them share the same pipeline:

- **Parsing:** The Modelica model is parsed and transformed into an Abstract Syntax Tree.
- **Flattening:** All object oriented structures are simplified. All array variables are transformed into scalar variables (de-vectorization) and all for-loops are transformed into a list of scalar equations (loop unrolling).
- **Matching:** Each scalar variable is matched with the scalar equation from which its value will be determined.
- **SCC resolution:** Algebraic loops are found and solved, where possible.
- **Scheduling:** Every scalar equation is explicitated and ordered accordingly to their mutual dependencies, so that the system can be sequentially simulated.
- **Lowering:** The executable code that simulates the model using the specified method and parameters is generated.

The use of de-vectorization and loop unrolling during the flattening stage greatly deteriorates the performance of the generated code.

2.4. MARCO Compiler

To address this problem, in the past years our research group (the PoliMi Modelica Working Group) has introduced a new Modelica compiler called MARCO (Modelica Advanced Research COmpiler) with performance and scalability as its main goals.
MARCO is based on the LLVM compiler framework and makes use of MLIR with a custom dialect, called Modelica dialect, as an intermediate representation (IR) of the internal state during the optimization passes [4]. The main feature of this compiler is that it does not make use of de-vectorization and loop unrolling during the flattening phase. The compilation pipeline of MARCO, similar to the one of a generic compiler, can be partitioned in three phases: front-end, middle-end and back-end.
In the front-end, the AST is built starting from the flattened source Modelica file. Then, the MLIR IR of the model is built starting from this AST. The middle-end of MARCO is where several optimization passes are performed, with the objective of solving the model, similarly to the previously mentioned compilers. The main difference is that MARCO, for the matching phase, uses a new algorithm that can preserve for-loops and array structures of the variables as much as possible [1]. At the end of this pass, every vector equation is matched with part of the vector variable it determines. Finally, the Modelica IR can transformed into LLVM-IR, which must in turn be translated by the back-end into assembly code suitable for the desired target architecture.

3. Design and Implementation

To add support for a DAE solver, it is necessary to generate the required code from the equations composing the model. This code will provide the DAE solver with the necessary data to compute the evolution of the state of the system or part of it.

3.1. Raw DAE Mode

The most simple way of using a DAE solver like IDA is to provide the entire DAE system to the solver in the form of equation 1. This way the system $\boldsymbol{F}$ will be directly solved with an implicit method like Newton's, and an integration algorithm such as the BDF method.
The main advantage of this approach is that the computation of derivatives and the integration procedure is done in one single pass. Furthermore, this is done with an implicit algorithm, thus solving implicitly both implicit algebraic equations and the implicit integration method. Another advantage is that we could skip entirely some stages of the compiler, such as the matching, the SCC resolution and the scheduling.
The disadvantage of this method, on the other hand, is that it scales poorly. Most of real-world phenomena usually only contain a small amount, between 5% and 20%, of differential equations and algebraic implicit equations. The rest of them are only alias equations or linear equations that depend on few variables and can be solved with a simple series of assignments.

3.2. Causalized DAE Mode

We want to address the problem of scalability. This can be achieved by combining the interesting aspects of the DAE-to-ODE translation and the Raw DAE method. The latter is interesting because it solves in one single pass the implicit stiff integration algorithm and the implicit algebraic equations. The former is also interesting because, since the amount of differential and implicit algebraic equations is small in proportion to the whole system, the computed BLT incidence matrix is "almost" an LT matrix.
This mixed approach can be implemented by selecting and exposing to the DAE solver only the BLT blocks that need such a solver to be computed, while hiding the remaining ones. As a result of this selection, we will divide all variables into two categories: trivial variables, which can be solved with a series of assignments, and non-trivial variables, which must be computed by a DAE solver.
Starting from the DAE system $\boldsymbol{F}$ of equation 1, we can unpack the vector of algebraic variables as $\boldsymbol{v} = [\boldsymbol{s}\,\boldsymbol{w}]$, where $\boldsymbol{s}$ are the trivial variables and $\boldsymbol{w}$ are the non-trivial algebraic variables. We can now split the system $\boldsymbol{F}$ as follows:

$$\begin{cases} \hat{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{w}(t), \boldsymbol{u}(t), t) = 0 \\ \boldsymbol{s}(t) = \tilde{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \end{cases} \tag{3}$$

where $\hat{\boldsymbol{F}}$ represent the implicit subsystem of the original model that must be solved with a DAE

solver, while $\tilde{\boldsymbol{F}}$ is the explicit subsystem that can be trivially solved. This way all the trivial variables $\boldsymbol{s}$ are hidden from IDA.
To transform the original systems $\boldsymbol{F}$ into the system in the form of equation 3, we must perform the following operations. First we must identify the vector $\boldsymbol{y} = [\dot{\boldsymbol{x}}\,\boldsymbol{w}]$ that must be passed to IDA inside the MARCO pipeline. Then, we must remove the dependencies from $\boldsymbol{s}$ inside the equations that must be passed to IDA. Finally, we must create and initialize the data required by the external DAE solver in order to compute the subsystem $\hat{\boldsymbol{F}}$.

3.3. Non-Trivial Variables Identification

The non-trivial variables that must be solved with IDA belong to one of these three categories:

- Algebraic equations belonging to an unsolved algebraic loop.
- Differential equations.
- Implicit algebraic equations.

The first ones can be found during the SCC resolution pass. The second category can then be searched through a linear scan of the equations, marking as non-trivial all of those matched with a derivative variable. Finally, the third category is found by trying to explicitate all equations with respect to the variable they are matched with. If this is not possible, the variable and the equation are marked as non-trivial.
Afterward, the scheduling procedure consists in obtaining the topological ordering of the DAG of the system, where each node is an equation or a SCC and edges represent the dependencies from variables. At the end of this pass, we will have obtained the final BLT incidence matrix (IM) of the system.

3.4. Trivial Variable Substitution

Once obtained the final BLT IM of the original system, we must extrapolate from it the subsystem that must be computed by the DAE solver. This can be done by recursively substituting all trivial variable accesses inside non-trivial equations with the right-hand side of the explicitated equation that is matched with such variables.
After applying this algorithm, all non-trivial equations are now independent from trivial variables. For this reason, we can reorder the IM of the system so that all non-trivial equations come before the trivial equations. Now that the subsystem $\hat{\boldsymbol{F}}$ that requires a DAE solver has been extrapolated from the original model, we can use its equations to create all the data required by IDA to compute the behavior of that subsystem.

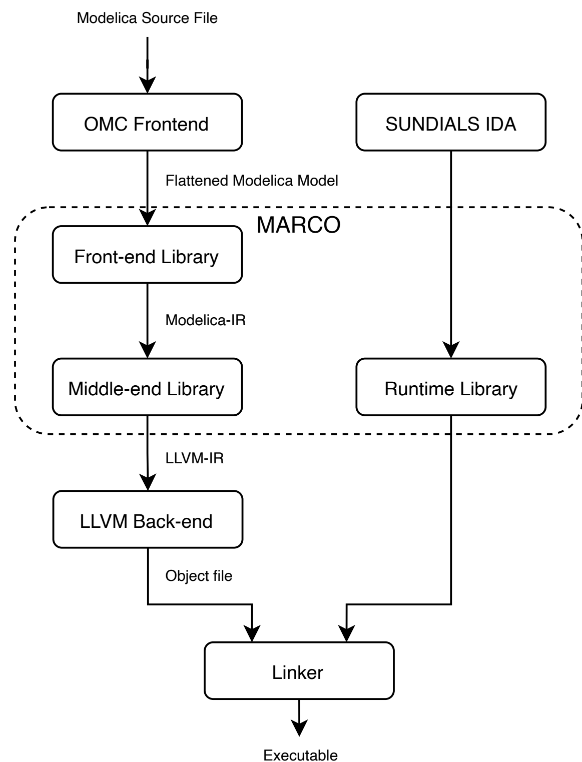

Figure 1: MARCO pipeline with IDA

3.5. Sundials IDA Integration

SUNDIALS IDA is the DAE solver chosen for the purposes of this project. This library was inserted inside the already available runtime library of MARCO. The new layout of MARCO is shown in figure 1.
A second MLIR dialect, called IDA dialect, has been introduced inside MARCO to interact with the external IDA library. This dialect depends on the already implemented Modelica dialect and it allows to write a more readable and maintainable IR during the various transformation passes of the middle-end. Most of the new operations introduced consist of simple wrappers around calls to the IDA runtime library. Among the other operations, two in particular are used to build parametric callback functions that compute the residual error and the Jacobian matrix of the subsystem $\hat{\boldsymbol{F}}$ starting from the non-trivial vector equations of the Modelica dialect.
Inside the middle-end of MARCO, after the model has been solved as described in the previ-

ous section, the last pass that needs to be performed is the introduction of calls to the IDA library inside the simulation program. A first set of calls must be inserted during the initialization of the simulation which allocate and provide to IDA the required information about the given subsystem, such as the number of variables and the aforementioned callback functions. Information about the simulation, such as start and end time, relative and absolute tolerances, can also be provided by a user. Then, during the main loop of the simulation, a single call to IDA must be performed at each iteration so that the DAE solver can compute the state of the system at the next time instant. Finally, at the end of the simulation, all the used data is deallocated after, eventually, printing some statistics related to the simulation.

4. Experimental Results

For testing our work, the benchmarking suite HiPerMod was used. This suite, developed by the PoliMi Modelica Working Group, was specifically made for testing the scalability of Modelica. We evaluate the correctness of the results, compilation time, simulation time, and resource usage of both compilation and simulation.

4.1. Thermal Chip DAE Model

For the experiments we employ the ThermalChipDAE model, which performs a thermal simulation like a finite element analysis. The Modelica source file describes a cube-shaped chip made of silicon, where a constant power is continuously applied, starting from an initial time t_0, on half of the bottom surface of the chip. The temperature across the volume of the cube is discretized into a three-dimensional matrix.The model was simulated with OMC, MARCO and C++ using different discretization parameters, doubling the number of state variables at each iteration, from $t_0 = 0.0$ seconds to $t_n = 1.0$ seconds.

4.2. Result Correctness

For what concern correctness, the tolerance parameters used by IDA are strictly related to the precision of the result since they determine the acceptable error of the simulation. For this reason, the three simulations were performed using a default value of 10^{-6} for both tolerances. Then, the simulation outputs were compared

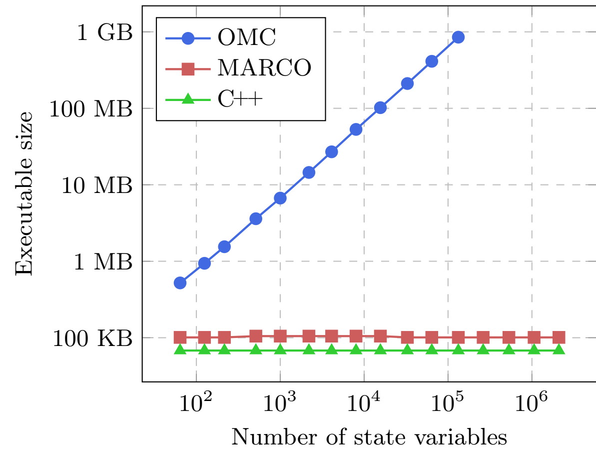

Figure 2: Binary size comparison

with the output of a simulation compiled with OMC using tolerances set to 10^{-10}. This way we can evaluate the errors made by MARCO and C++ through a comparison with the error that OMC makes among its two simulations.
For every simulation, the maximum relative error was between 7×10^{-7} and 3×10^{-6} K, while the maximum absolute error was between 3×10^{-4} and 9×10^{-4} K. This is because the error accumulates through the simulation at every step, eventually becoming greater than the tolerance parameter. In any case, the errors made by MARCO and C++ were the same as those made by OMC. For this reason the resulting errors do not depend on the compiler but only on the DAE solver.

4.3. Binary Size

As can be seen in figure 2, the sizes of the executables produced by MARCO and C++ are both constant. Instead, the OMC binary size grows linearly with the number of variables used, reaching almost 1 GB of size. This is due to the complete de-vectorization of variables and the loop unrolling that OMC performs at compile time as explained in section 2.3.

4.4. Compilation Time

Figure 3 shows the times required to create the three simulation executables. It is clear that the objective of compiling the model in constant time with respect to the number of scalar equations has been reached. The compilation time of OMC, on the other hand, grows exponentially with respect to the number of scalar variables. This is, again, caused by de-vectorization and

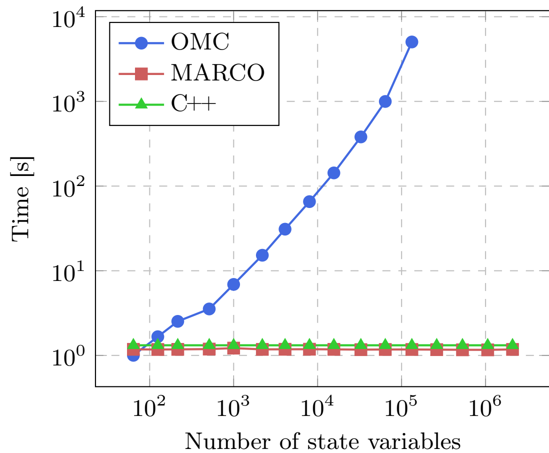

Figure 3: Compilation time comparison

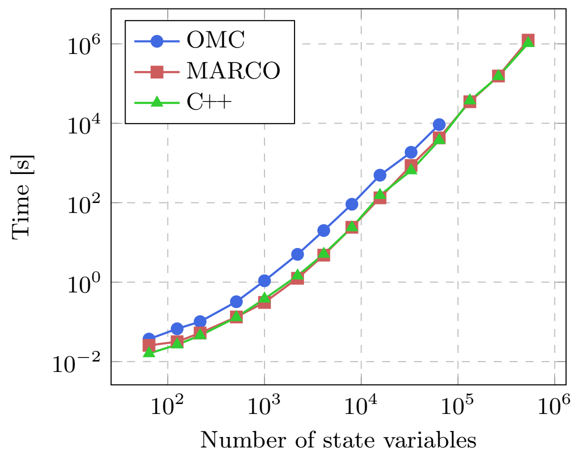

Figure 4: Simulation time comparison

loop unrolling.

4.5. Simulation Time

The performance improvement of MARCO with respect to OMC for what concern the simulation time were also benchmarked and the results are shown in figure 4. All simulations scale quadratically with respect to the number of states. This is due to the behavior of the DAE solver that must compute several $m \times m$ matrices, where m is the number of scalar non-trivial variables computed by IDA. That said, we can notice that the executable produced by MARCO is around twice as fast as the one produced by OMC, like the C++ one, thus still showing a significant advantage. We can therefore conclude that MARCO is able to obtain the optimal simulation of a model using a DAE solver and that the bottleneck is in the solver itself.

5. Conclusions

In this document we have presented the usefulness and the many difficulties of simulating real-world phenomena by means of differential and algebraic equations. To tackle this problem, a new Modelica compiler called MARCO, currently with limited capabilities, has been introduced. The main objective of this new compiler is to preserve array structures and for loops inside the simulation executable, which is something that the currently available Modelica compilers do not do. As the purpose of this thesis, the variable step size DAE solver called IDA has been introduced into MARCO, while maintaining its original objective.
Both a simplified “raw” implementation of the DAE solver and a more efficient design have been presented. The latter aims at using the DAE solver only for those variables that strictly require it. Such variables are the ones contained in algebraic loops, implicit algebraic equations and differential equations: the MARCO pipeline has been modified to accommodate these needs so that these variables can be identified. Then, through the introduction of a new MLIR dialect, a new transformation pass has been introduced that takes all this information, creates the functions needed by IDA and the appropriate calls to the external runtime library.
Finally, the usefulness of this approach, compared to other compilers, has been evaluated through several tests.

References

[1] Massimo Fioravanti. M.A.R.C.O.: an experimental high-performance Modelica compiler for large scale systems. Master’s thesis, Politecnico di Milano, 2020.

[2] Alan Hindmarsh et al. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 2005.

[3] Sven Erik Mattsson et al. Physical system modeling with Modelica. *Control Engineering Practice*, 1998.

[4] Michele Scuttari. Design and implementation of a Modelica compiler with MLIR and LLVM. Master’s thesis, Politecnico di Milano, 2021.