Clever DAE: Compiler Optimizations for Digital Twins at Scale

Michele Scuttari
Politecnico di Milano
Milan, Italy
michele.scuttari@polimi.it

Nicola Camillucci
Politecnico di Milano
Milan, Italy
nicola.camillucci@mail.polimi.it

Daniele Cattaneo
Politecnico di Milano
Milan, Italy
daniele.cattaneo@polimi.it

Giovanni Agosta
Politecnico di Milano
Milan, Italy
giovanni.agosta@polimi.it

Francesco Casella
Politecnico di Milano
Milan, Italy
francesco.casella@polimi.it

Stefano Cherubin
Edinburgh Napier University
Edinburgh, United Kingdom
s.cherubin@napier.ac.uk

Federico Terraneo
Politecnico di Milano
Milan, Italy
federico.terraneo@polimi.it

ABSTRACT

Modeling and simulation are fundamental activities in engineering to facilitate prototyping, verification and maintenance. Declarative modeling languages allow to simulate physical phenomena by expressing them in terms of Differential and Algebraic Equations (DAE) systems. In this paper, we focus on the problem of generating code for performing the numerical integration of the model equations, and in particular on the overhead introduced by external numerical solver libraries. We propose a novel methodology for minimizing the amount of equations which require to be solved through an external solver library, together with the number of computations that are required to computed the Jacobian matrix of the system. Through a prototype LLVM-based compiler, we demonstrate how this approach achieves a linear speed-up in simulation time with respect to the baseline.

CCS CONCEPTS

• **Software and its engineering** → *Compilers*; • **Computing methodologies** → *Modeling and simulation.*

KEYWORDS

Modelica, compiler, simulation

ACM Reference Format:
Michele Scuttari, Nicola Camillucci, Daniele Cattaneo, Giovanni Agosta, Francesco Casella, Stefano Cherubin, and Federico Terraneo. 2023. Clever DAE: Compiler Optimizations for Digital Twins at Scale. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587135.3589945

CF '23, May 9–11, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0140-5/23/05…$15.00
https://doi.org/10.1145/3587135.3589945

1 INTRODUCTION AND RELATED WORKS

The changes induced by the so-called "Fourth Industrial Revolution", or, more commonly, *Industry 4.0* have driven a renewed interest in modeling and simulation technologies. In particular, the concept of *digital twin* [3, 6, 15, 16] promises to enable a wide range of design and maintenance tasks on real-world physical (and cyber-physical) systems such as cars, planes, buildings and power-distribution networks, by employing large scale simulations of these very complex systems. Digital twins model physical systems and phenomena that can be expressed in terms of Differential and Algebraic Equations (*DAE*). However, developing software models for large scale systems using general purpose programming paradigms is an inefficient and error-prone operation. To address this gap, declarative modeling languages have been introduced. Such languages allow to create a digital twin by writing directly the DAEs that describe the behavior of the physical system being modeled. Modelica [17] is one such language that has gained a significant traction.

The fundamental construct of declarative modeling languages is the *equation*. However, equations cannot be directly mapped onto statements of programming languages, as this mapping depends on the chosen numerical integration algorithm. Several such algorithms have been proposed, with different trade-offs. Explicit methods such as Forward Euler do not require to solve implicit equations systems but have limited numerical stability, often requiring short integration steps. Implicit methods such as the Backward Euler method suffer from complementary trade-offs, while variable step methods can dynamically adapt the numerical integration step to keep the solution error below a given tolerance [8]. From the compiler perspective, the numerical integration method can be either integrated in the produced simulation executable — the obvious choice for simple techniques such as explicit methods — or reside in an external library — preferred for implicit and variable step methods [14]. However, when compiling large-scale models featuring millions of equations, the amount of interface code required by such solver libraries to read the structure of the model quickly saturates all computing resources. Indeed, methods involving the computation of a Jacobian matrix scale quadratically in time and memory complexity with the number of equations [2]. Even

if we were able to bypass this bottleneck by employing virtually-unlimited resources, the run time would constitute the next blocker.

In this work, we address the scalability problem by reducing the amount of computational resources — time and space — required by large scale systems. More specifically, we introduce a partitioning algorithm which aims to use the external solver only for a minimal set of variables and equations, alongside with two other optimizations that reduce the computational effort required to generate the Jacobian matrix. We demonstrate the advantages of our approach using the IDA solver library — part of the SUNDIALS suite [14] — and, in particular, its implementation of a variable order and variable step size BDF algorithm. Nonetheless, the same principles can be applied on other algorithms and solver libraries.

The nearest works to ours are those on the implementation of equation-based modeling languages. There is a great variety of different languages and tools in this class, both application-specific and general-purpose. Modelica and its competitors (gPROMS [7], Simscape [19] and Omola [5]) fall in the latter class. Considering Modelica compilers, OpenModelica [11] is the only open source option, whereas Dymola [9] and JModelica [4] are proprietary. Considering the optimizations presented in this work, they are not specific with respect to the compiler employed, although it is impossible to say whether they are adopted in either Dymola or JModelica. Given the known performance limitations of OpenModelica when dealing with large models [1, 10], any performance benefit it might get would still be overshadowed by the large penalties imposed by the processing of such large models.

2 BACKGROUND

In this section, we review the concepts needed to understand the specificities of the Modelica language and its application domain, as well as the current state of the art in the compilation of Modelica code.

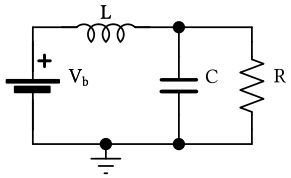
2.1 DAE Models

In the most generic form, a system of DAEs can be written as shown in Equation 1.

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

The state variables vector $\boldsymbol{x}$, the derivative variables vector $\dot{\boldsymbol{x}}$, and the algebraic variables vector $\boldsymbol{v}$ represents the unknowns of the system, while the input vector $\boldsymbol{u}$ is always known.

In order to integrate a DAE system, the equations must be *causalized*, in other words translated into a series of assignments that compute the state variables at each time-step. To do so, each equation is made explicit with respect to a *matched* variable which could either be an algebraic unknown or the derivative of a state variable, which must be unique for every equation. The whole set of equations is then reordered in a way such that its Incidence Matrix (IM) — i.e. a matrix which describes which variables appear in every equation — becomes lower triangular (LT). If this is possible, then all unknowns can be computed for the current time step, but in the general case the IM can only be made Block Lower Triangular (BLT), that is a matrix whose diagonal is composed of many $m_i \times m_i$ blocks, with $m_i \leq n$, each of which having possibly non-zero non-diagonal elements. In this case the blocks correspond to system of equations that have to be solved using numerical methods [8].



(a) Circuit

	i_R	i_C	$\dot{i_L}$	$\dot{V}$
eq_1	1	0	0	0
eq_2	1	1	0	0
eq_3	0	0	1	0
eq_4	0	1	0	1

(b) Incidence matrix

Figure 1: RLC Circuit

To better understand this process, let us consider as an example the simple electrical circuit shown in Figure 1a and described by the DAE system shown in Equation 2, whose components have already been matched and ordered according to their dependencies. Figure 1b represents the IM: given the cell at (i, j) position, where i and j represent the row and the column respectively, its value contains the 1 value if variable j appears in equation i. Being the matrix LT, the equations can be translated into assignments to be performed at each time step.

$$\begin{cases} eq_1: & i_R = V/R \\ eq_2: & i_C = i_L - i_R \\ eq_3: & \dot{i_L} = (V_b - V)/L \\ eq_4: & \dot{V} = i_C/C \end{cases} \tag{2}$$

Note that the state variables V and i_L are assumed to be known, as at the first step of the numerical integration their initial values are used, while the value computed at the previous step is used by the numerical integration algorithm to compute the next one using the computed derivatives.

Integration mechanisms such as the ones described are called DAE solvers and, as anticipated in Section 1, in this document we will focus our attention on a solver software library called IDA implementing a variable step BDF solver for DAE systems. To use this solver, the compiler needs to produce code to compute the residual functions and Jacobian matrix of the system to be integrated, which are then used by the BDF method.

Given an equation, its residual function is the difference between its right-hand side and left-hand side values. The Jacobian matrix is instead a square matrix defined as in Equation 3.

$$\boldsymbol{J} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial F}{\partial x_1} & \cdots & \frac{\partial F}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^T F_1 \\ \vdots \\ \nabla^T F_n \end{bmatrix} = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix} \tag{3}$$

Each of the n rows represents an equations of the system, while each of the n columns represents a variable. The element at position (i, j) represents the partial derivative of the residual function of equation i with respect to variable j. It should be noted that the steps here presented are only applicable to index-1 DAE models, that is models with a non-singular Jacobian matrix. Higher index systems are outside the scope of this work.

2.2 Automatic differentiation

The computation of the Jacobian matrix of the system requires the computation of the partial derivatives, which can be performed

through *automatic differentiation* (AD). There are multiple algorithms for performing AD, in this work we only consider the forward accumulation algorithm (*forward AD*).

In forward AD, the first step is to establish the independent variable with respect to which the differentiation is performed. To this end, every variable is associated with a *seed* derivative, which is set to 1 if we are differentiating with respect to that variable, and to 0 otherwise. The algorithm proceeds by substituting the derivative of inner functions by recursively applying the chain rule. The recursion ends when encountering a variable whose derivative is the previously set seed [13].

As an example, let us consider the following function:

$$f(x_1, x_2) = x_1 x_2 + \sin x_1$$

To compute its derivative with respect to x_1, first we decompose the computation of f in elementary operations, whose results are assigned to temporary variables w_i. Now, we compute the derivative $\dot{w}_i$ for each temporary variable through the chain rule. This process is shown in the following table.

Original operations	Derivative operations
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = \dot{w}_1 \cdot w_2 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

2.3 The Modelica Language

Modelica [17] 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.

A model example can be seen in code listing 1, which describes the heat transfer in a wire. It is important to note how, differently from imperative languages, the equal sign = does not represent an assignment operation nor it states causality among variables, but rather represents just the declaration of an equation. Another notable feature is the possibility of declaring parametric multidimensional arrays, for example to discretize the length of the wire. This is effectively equivalent to a declaration of several scalar variables at once. Last but not least, for-loops within Models do not represent control flow inside the simulation, but are used to express multiple similar equations in a compact form.

The two most used Modelica compilers are currently OpenModelica Compiler (OMC) [11] and Dymola [9]. Both of them share the same pipeline: (1) **Parsing and Flattening:** The Modelica source code is parsed and transformed into an Abstract Syntax Tree. Then, all object oriented structures and other syntactic sugar such as for-loops are lowered. (2) **Matching:** Each scalar variable is assigned to a scalar equation that will update its value. (3) **SCC resolution:** Algebraic loops are found and solved, when possible. (4) **Scheduling:** Every scalar equation is made explicit with respect to its matched variable and the whole list of equations is ordered accordingly to their mutual dependencies, so that the system can be sequentially simulated. (5) **Lowering:** The executable code that simulates the model using the specified method and parameters is generated.

```
model ThermalWire
  parameter Real area = 0.0005^2 * 3.14;
  parameter Real length = 0.1;
  parameter Real conductivity = 401;
  parameter Real specificheatcapacity = 385;
  parameter Real density = 8960;
  parameter Real g = conductivity * area / length;
  parameter Real c = specificheatcapacity *
      density * area * length;
  parameter Real Thigh = 400 + 273.15;
  parameter Real Tlow = 20 + 273.15;
  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;
```

Listing 1: Heat transfer in a 1D wire

With this knowledge of modeling languages it is now possible to introduce the problem addressed in this paper and describe the proposed solution.

3 PROPOSED SOLUTION

To integrate a model using a DAE solver, it is necessary to generate the required code to inform the solver about the equations composing the system. The easiest approach is to feed the entire set of equations and variables to the solver. This reduces the amount of analyses the compiler needs to perform: all the mathematical aspects would be entirely delegated to the external tool. The downside of this approach is its poor scalability. Most real-world phenomena only contain a small amount of differential equations and algebraic implicit equations while the rest are typically alias definitions or linear equations that can be eliminated [18].

To address the problem of scalability, we can take into account two different aspects. The first regards the identification of the BLT blocks that are strictly required by the DAE solver, to make it operate on a reduced system: by reducing the number of variables and equations, also the memory and time footprint can be reduced. The second is to optimize the runtime computation of the Jacobian matrix, which can be achieved by analysing the structure of equations and by taking advantage of mathematical properties coupled with forward AD.

3.1 Computation of the Reduced System

We conceptually divide variables into two categories: *trivial* and *non-trivial* variables. The former contains the variables whose matched equations can be made explicit by the compiler through proper algebraic manipulations. The latter contains the derivative variables of the system, which must be handed over to the DAE solver, together with the variables that have been matched with implicit equations or equations belonging to unsolved cycles.

Starting from the DAE system $\boldsymbol{F}$ of Equation 1, the vector of algebraic variables can be unpacked as $\boldsymbol{v} = [\boldsymbol{s}\,\boldsymbol{w}]$, where $\boldsymbol{s}$ are the trivial variables and $\boldsymbol{w}$ are the non-trivial variables. The system $\boldsymbol{F}$ can be now split according to Equation 4.

$$\begin{cases} \hat{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{w}(t), \boldsymbol{u}(t), t) = 0 \\ \boldsymbol{s}(t) = \tilde{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \end{cases} \tag{4}$$

$\hat{\boldsymbol{F}}$ represents the implicit subsystem of the original model that must be solved with a DAE solver, while $\tilde{\boldsymbol{F}}$ is the subsystem that can be trivially solved.

To transform the original systems $\boldsymbol{F}$ into the system in the form of Equation 4, two steps must be performed:

- Identification of the vector $\boldsymbol{y} = [\dot{\boldsymbol{x}}\,\boldsymbol{w}]$, representing the variables handled by the DAE solver. While doing this operation, also the respective matched equations must be identified to separate the two systems.
- Removal of the dependencies from $\boldsymbol{s}$ inside the equations handled by the DAE solver.

The first task can be performed through a linear scan of the equations, marking as non-trivial the ones that are matched with a derivative variable. Implicit equations can be also identified while performing the search of differential equations. Unsolved algebraic loops have already been identified during the SCC resolution process, and so they can be added without additional analyses.

Afterwards, the externalized equations must be transformed such that they do not contain any dependency from the trivial variables and thus their execution can be grouped together. For example, consider the example system in Equation 5 and its incidence matrix (IM) shown in Figure 2. It is clear how the externalized equations – marked in red – are interleaved with the others – marked in green – making each subsystem dependent on the other.

$$\begin{cases} s_3 = f_6'(\boldsymbol{x}, \boldsymbol{u}, t) \\ f_3(\boldsymbol{x}, s_3, \boldsymbol{w}_8, \boldsymbol{u}, t) = 0 \\ s_1 = f_8''(\boldsymbol{x}, s_3, \boldsymbol{u}, t) \\ s_7 = f_2'(\boldsymbol{x}, s_1, \boldsymbol{w}_8, \boldsymbol{u}, t) \\ f_4(\boldsymbol{x}, s_3, \boldsymbol{w}_4, \boldsymbol{w}_5, s_7, \boldsymbol{u}, t) = 0 \\ f_1(\boldsymbol{x}, s_1, \boldsymbol{w}_4, \boldsymbol{w}_5, \boldsymbol{w}_8, \boldsymbol{u}, t) = 0 \\ s_2 = f_9'(\boldsymbol{x}, \boldsymbol{w}_5, s_7, \boldsymbol{w}_8, \boldsymbol{u}, t) \\ f_7(\boldsymbol{x}, \dot{\boldsymbol{x}}, s_2, s_3, \boldsymbol{w}_5, s_7, \boldsymbol{u}, t) = 0 \\ s_6 = f_5'(\boldsymbol{x}, \dot{\boldsymbol{x}}, s_1, \boldsymbol{w}_4, \boldsymbol{w}_8, \boldsymbol{u}, t) \end{cases} \tag{5}$$

This overlap can be solved by substituting the usages of the trivial variables inside the externalized equations with their equivalent expressions. This explains the need for the trivial variables to be matched with equations that can be made explicit, and thus the delegation of implicit equations and unsolved cycles to the DAE solver.

	s_3	$\boldsymbol{w}_8$	s_1	s_7	$\boldsymbol{w}_5$	$\boldsymbol{w}_4$	s_2	$\dot{\boldsymbol{x}}$	s_6
f_6'	1	0	0	0	0	0	0	0	0
f_3	1	1	0	0	0	0	0	0	0
f_8''	1	0	1	0	0	0	0	0	0
f_2'	0	1	1	1	0	0	0	0	0
f_4	1	0	0	1	1	1	0	0	0
f_1	0	1	1	0	1	1	0	0	0
f_9'	0	1	0	1	1	0	1	0	0
f_7	1	0	0	1	1	0	1	1	0
f_5'	0	1	1	0	0	1	0	1	1

Figure 2: Example of incidence matrix after scheduling

	$\boldsymbol{w}_8$	$\boldsymbol{w}_5$	$\boldsymbol{w}_4$	$\dot{\boldsymbol{x}}$	s_3	s_1	s_7	s_2	s_6
f_3'	1	0	0	0	0	0	0	0	0
f_4'	0	1	1	0	0	0	0	0	0
f_1'	0	1	1	0	0	0	0	0	0
f_7'	0	1	0	1	0	0	0	0	0
f_6'	0	0	0	0	1	0	0	0	0
f_8''	0	0	0	0	1	1	0	0	0
f_2'	1	0	0	0	0	1	1	0	0
f_9'	1	1	0	0	0	0	1	1	0
f_5'	1	0	1	1	0	0	0	0	1

Figure 3: Example of incidence matrix after substitution of trivial variables

Once the replacements have been performed and the scheduling process re-executed, the resulting IM will have the form shown in Equation 6, with A being BLT and D being LT.

$$IM = \begin{bmatrix} A & 0 \\ C & D \end{bmatrix} \tag{6}$$

Considering again the example system in Equation 5, the resulting IM is shown in Figure 3.

3.2 Jacobian Matrix Computation

To compute the Jacobian matrix of the system, as required by the variable-step BDF method discussed in Section 2.1, the compiler needs to generate the derivative of the system equations w.r.t. all the variables handled by DAE solver. As anticipated in Section 2.2, we employ automatic differentiation to compute them.

Reduction of computed derivatives. More in detail, forward accumulation AD has been preferred over backward mode AD, despite the former having a greater computational complexity [12]. In fact, by collecting the set of variables accessed by each equation, it is possible to obtain compile-time knowledge about which independent variables lead to zero-valued derivatives for that equation, which therefore do not need to be computed. As real-world systems often are highly sparse — as it can be seen even just from the example IM in Figure 2 — these zero-valued derivatives form the great majority of the cells in the Jacobian matrix. Since forward accumulation AD only computes a single derivative at once, it is possible to skip known-zero derivatives, thus introducing the first optimization which reduces both compilation and simulation times.

On the other hand, backward mode AD cannot compute a subset of the partial derivatives for a given equation, thus always requiring the computation of the entire Jacobian matrix unconditionally.

Optimization of AD seeds. Moreover, the IDA solver uses an approximation of the Jacobian matrix given by Equation 7.

$$\boldsymbol{J} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} + \alpha \frac{\partial \boldsymbol{F}}{\partial \dot{\boldsymbol{x}}} \tag{7}$$

By starting from Equation 7 and defining the new operator $\boldsymbol{\partial^{\alpha}}$ as in Equation 8, it is possible to obtain the equality given by Equation 9.

$$\boldsymbol{\partial^{\alpha}} = \frac{\partial}{\partial x} + \alpha \frac{\partial}{\partial \dot{x}} \tag{8}$$

$$\boldsymbol{\partial^{\alpha}} \left(F\left(G\left(x, \dot{x}\right)\right)\right) = \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \boldsymbol{\partial^{\alpha}} \left(G\left(x, \dot{x}\right)\right) \tag{9}$$

Proof is given by the list of equalities in Equation 10.

$$\begin{aligned}
&\boldsymbol{\partial^{\alpha}} \left(F\left(G\left(x, \dot{x}\right)\right)\right) = \\
&= \left(\frac{\partial}{\partial x} + \alpha \frac{\partial}{\partial \dot{x}}\right) \left(F\left(G\left(x, \dot{x}\right)\right)\right) \\
&= \frac{\partial F\left(G\left(x, \dot{x}\right)\right)}{\partial x} + \alpha \frac{\partial F\left(G\left(x, \dot{x}\right)\right)}{\partial \dot{x}} \\
&= \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \frac{\partial G\left(x, \dot{x}\right)}{\partial x} + \alpha \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \frac{\partial G\left(x, \dot{x}\right)}{\partial \dot{x}} \\
&= \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \left(\frac{\partial G\left(x, \dot{x}\right)}{\partial x} + \alpha \frac{\partial G\left(x, \dot{x}\right)}{\partial \dot{x}}\right) \\
&= \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \left(\frac{\partial}{\partial x} + \alpha \frac{\partial}{\partial \dot{x}}\right) \left(G\left(x, \dot{x}\right)\right) \\
&= \dot{F}\left(G\left(x, \dot{x}\right)\right) \cdot \boldsymbol{\partial^{\alpha}} \left(G\left(x, \dot{x}\right)\right)
\end{aligned} \tag{10}$$

In the context of AD, this can be performed at runtime as a single step rather than two by setting to 1 the seed of the state variable x and to α the seed of its derivative $\dot{x}$.

4 EXPERIMENTAL EVALUATION

To measure the effectiveness of our approach, we use a Modelica model describing a cube-shaped chip of silicon, where a constant power is continuously applied on half of its bottom surface. The temperature across the volume of the chip is discretized into a parametrized three-dimensional matrix.

The results are obtained by using an in-house prototype compiler based on the LLVM infrastructure. Compilation time, executable size and simulation time are evaluated with respect to five possible configurations: **Raw DAE** no optimization; **Reduced system opt** reduced system computation only; **Zero der opt** possibly non-zero partial derivatives computation only; **Alpha opt** AD seeds optimization only; **Clever DAE** all optimizations.

All the tests are performed on a machine with the following specifications: **OS**: Ubuntu 20.04, **CPU**: 10-core Intel Xeon E5-2650 v3 2.30GHz, **RAM**: 72 GB DDR3 2133 MHz, LLVM 16.0.0.

Compilation Time. Figure 4 shows the times required to apply the transformation pass generating the residual functions, the partial derivatives and the initialization code for the external solver. Although applying the presented optimizations increases the time required to perform this step of the compilation, this increase is compensated by the reduced simulation time and executable code size.

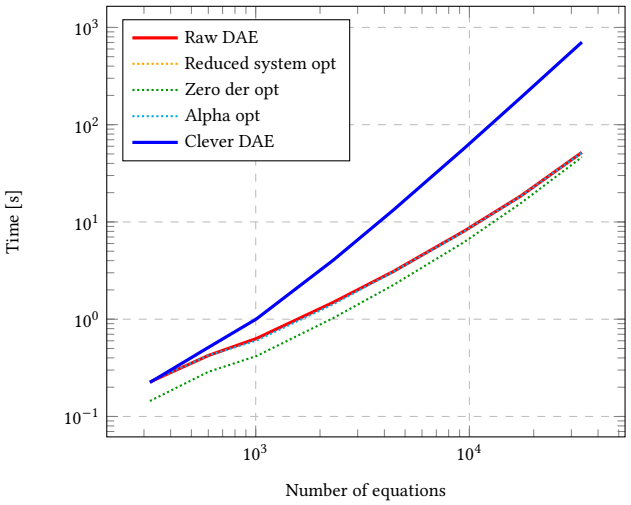


Figure 4: Compiler time required for the generation of the external solver handling code in the simulation

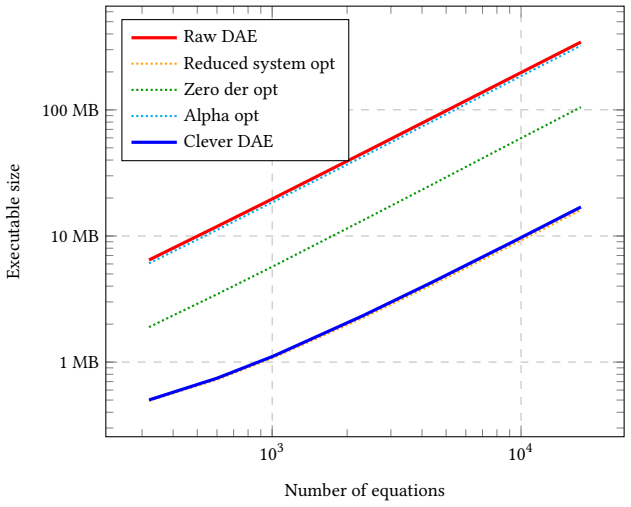


Figure 5: Binary size

The AD seeds optimization on the other hand does not impact compilation time in a noticeable way. Indeed, the only difference is that the compiler generates a single call instruction to a common derivative function, rather than two. Therefore, with the tested model, the compile-time complexity of the optimization is not significant.

Binary Size. Figure 5 shows the size of the generated simulation executable. The simulations generated with the Clever DAE method are ~ 20 times smaller with respect to the ones generated with the Raw DAE method. For what regards the individual optimizations, the situation resembles the one seen for compilation times: the main improvement is again given by the reduction of the externally processed set of equations, but the computation of always-zero partial derivatives also leads to a considerable ~ 70% reduction of the executable file size when considered on its own.

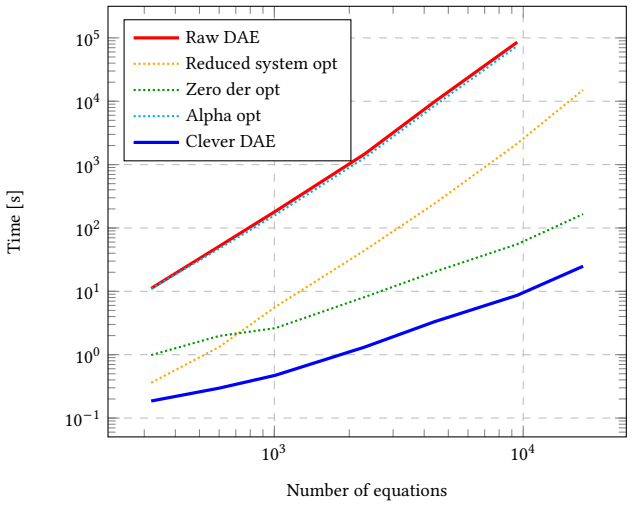


Figure 6: Simulation time

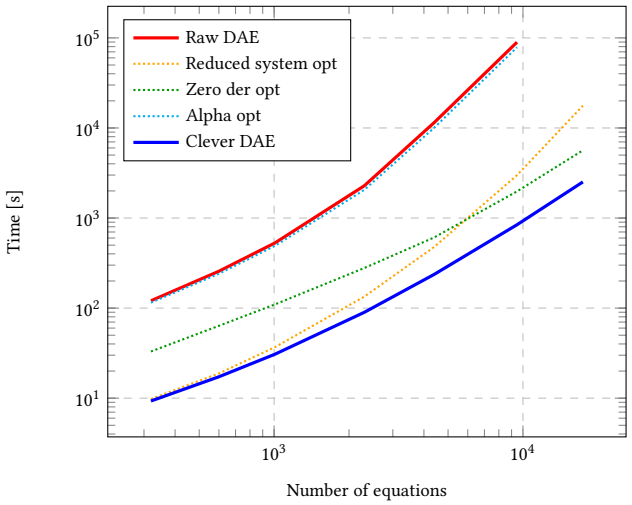


Figure 7: Time required to compile the model and run the simulation

Simulation Time. Figure 6 shows the time taken by the simulation to complete its execution. Considering the Clever and Raw DAE methods, the former outperforms the latter as the number of equations grows. With ∼ 300 equations there is already an improvement of ∼ 60 times, which increases to ∼ 9800 with roughly 10000 equations.

This improvement originates from the pre-computation of the always zero-valued partial derivatives, which allows to overcome the quadratic nature of the Jacobian matrix by just setting the few seeds – whose quantity is constant – needed for each row of the Jacobian matrix. On the other hand, the AD seeds optimization is negligible. This is caused by the specific features of the model considered for this benchmark, whose partial derivatives do not lead to common sub-expressions whose value can be reduced to a single runtime computation.

Time To Solution. In the context of modeling and simulation, it is common for models to be compiled and simulated only once. Figure 7 shows the time required to obtain the simulation data starting with the source code describing the system of equations.

As can be seen, the situation mimics the one seen for simulation times, with the Raw DAE method diverging from the Clever DAE.

5 CONCLUSIONS

This work represents an important step towards the adoption of large-scale DAE-based digital twins. More in detail, we have presented three possible optimizations to improve the simulation of DAE systems solved by using the IDA solver.

The first one consists in the determination of the minimal set of equations – and variables – that the solver needs to handle in order to produce a correct result.

The second and third ones enable an efficient computation of the Jacobian matrix, which is required by the BDF method used by the aforementioned solver. The former consists in the pre-computation of the always zero-valued partial derivatives, while the latter makes use of mathematical properties to allow the reduction of computations in case of sub-expressions that are common to both the derivatives with respect to the state and derivative variables.

The usefulness of these transformations has been evaluated through an highly scalable Modelica model, processed with a prototype compiler based on the LLVM infrastructure.

Even though compilation times increase slightly, our tests have shown significant simulation performance gains. In particular, simulation times are reduced throughout the whole tests in a linear way, obtaining an improvement up to ∼ 9800 times at ∼ 10000 equations. Binary sizes have also been reduced in a constant manner, obtaining a ∼ 70% improvement. To also improve compilation times, the proposed techniques could be combined with exploiting array structures by avoiding scalar expansion during the code generation process [1]. Although this is postponed to future works, combining these techniques has the potential to enable simulation of truly large scale systems.

ACKNOWLEDGMENTS

Michele Scuttari was supported by the Italian Ministry of University and Research and by Huawei Italy under the PNRR program (Piano Nazionale di Ripresa e Resilienza).

REFERENCES

[1] Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. 2019. Towards a High-Performance Modelica Compiler. (2019), 313–320. https://doi.org/10.3384/ecp19157313

[2] Giovanni Agosta, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. 2020. Towards a Benchmark Suite for High-Performance Modelica Compilers. In *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools* (Berlin, Germany) *(EOOLT '19)*. Association for Computing Machinery, New York, NY, USA, 21–24. https://doi.org/10.1145/3365984.3365988

[3] Panagiotis Aivaliotis, Konstantinos Georgoulias, Zoi Arkouli, and Sotiris Makris. 2019. Methodology for enabling digital twin using advanced physics-based modelling in predictive maintenance. *Procedia CIRP* 81 (2019), 417–422.

[4] Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. 2009. Jmodelica—an open source platform for optimization of modelica models. In *6th Vienna International Conference on Mathematical Modelling*.

[5] Mats Andersson. 1989. An object-oriented language for model representation. In *Proc. 2nd IEEE Control Systems Society Workshop on Computer-Aided Control System Design*. Tampa, FL, USA, 8–15.

[6] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. 2019. A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications. *IEEE Access* 7 (2019), 167653–167671. https://doi.org/10.1109/ACCESS.2019.2953499

[7] Paul I. Barton and Constantinos C. Pantelides. 1994. Modeling of combined discrete/continuous processes. *AIChE journal* 40, 6 (1994), 966–979.

[8] François E. Cellier and Ernesto Kofman. 2006. *Continuous system simulation*. Springer Science & Business Media, Berlin, Germany.

[9] Hilding Elmqvist. 1979. DYMOLA – A Structured Model Language for Large Continuous Systems. In *Proc. Summer Computer Simulation Conference*. Toronto, Canada.

[10] Massimo Fioravanti, Daniele Cattaneo, Federico Terraneo, Silvano Seva, Stefano Cherubin, Giovanni Agosta, Francesco Casella, and Alberto Leva. 2022. Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models. https://doi.org/10.48550/ARXIV.2212.11135

[11] Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. 2006. OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In *2006 IEEE Conf on Computer Aided Control System Design, 2006 IEEE Int'l Conf on Control Applications, 2006 IEEE Int'l Sym on Intelligent Control*. IEEE, 1588–1595.

[12] Andreas Griewank et al. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.

[13] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM.

[14] Alan Hindmarsh et al. 2005. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM Transactions on Mathematical Software (TOMS)* (2005).

[15] Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. 2020. A state-of-the-art survey of Digital Twin: techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing* 31, 6 (2020), 1313–1337.

[16] Mengnan Liu, Shuiliang Fang, Huiyue Dong, and Cunzhi Xu. 2021. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems* 58 (2021), 346–361. https://doi.org/10.1016/j.jmsy.2020.06.017 Digital Twin towards Smart Manufacturing and Industry 4.0.

[17] Sven Erik Mattsson et al. 1998. Physical System Modeling with Modelica. *Control Engineering Practice* (1998).

[18] Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. 2014. Integrated Debugging of Modelica Models. *Modeling, Identification and Control* 35, 2 (2014), 93–107. https://doi.org/10.4173/mic.2014.2.3

[19] The Mathworks, Inc. 2022 (latest version). Simscape Documentation. https://mathworks.com/help/physmod/simscape/.