MARCO: An Experimental High-Performance Compiler for Large-Scale Modelica Models

Giovanni Agosta[1] Francesco Casella[1] Daniele Cattaneo[1] Stefano Cherubin[2] Alberto Leva[1]
Michele Scuttari[1] Federico Terraneo[1]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
{name.surname}@polimi.it
[2]NTNU - Norwegian University of Science and Technology, Norway, stefano.cherubin@ntnu.no

Abstract

This paper introduces MARCO, a research compiler aimed at the efficient generation of efficient simulation code from a large-scale Modelica model. MARCO's design goals, requirements, and specifications are discussed in the paper, as well as the software architecture, the current development status, and a future development roadmap. The results of two test cases demonstrate MARCO's capability to handle non-trivial Modelica models with over 10 million equations very efficiently.

Keywords: Modelica, compiler construction, large scale models

1 Introduction

The Modelica Language, first introduced in 1997, has become a widespread standard in the field of system-level modelling and simulation, and is now supported by many different tools, both commercial and open source. For many years, the focus of Modelica tools was mainly to support the modelling of individual systems, such as a robot, a power plant, an air conditioning system, a heat pump, an aircraft, and so on. Such models are built by connecting heterogeneous sub-systems belonging to different physical domains (e.g., mechanical, thermal, electrical) and often require the efficient and robust solution of non-trivial systems of non-linear equations. However, their scale usually is quite limited, from a few hundred equations up to one or two hundred thousand equations. According to how flattening is described in the language specification (Modelica Association 2021), such models are transformed into a system of scalar equations involving scalar variables.

This approach has served the Modelica community well for about 25 years, but suffers from severe performance limitations in two cases. One is the case of models containing large array equations, e.g., stemming from the 2D or 3D discretisation of distributed-parameters systems. The other is the case of systems-of-systems with many repeated instances of the same basic components. In both cases, the typical workflow of today's Modelica tools turns out to be inefficient, particularly as regards the time required to generate the executable simulation code from the original Modelica source code, and also as regards the size of the generated code, which contains many repetitions of essentially the same lines of code. This issue was highlighted eight years ago in (Casella 2015), Section 2.6, but until now, no industrial-grade solutions have been developed to overcome this problem.

This issue hampers the use of Modelica for effectively modelling systems-of-systems and large-scale, smart, distributed systems of all kinds (e.g., smart grids, smart neighbourhoods, and IoT systems in general). The Modelica language perfectly suits the task, as it can conveniently describe structured multi-domain cyber-physical systems. Still, tools are not up to the task when the size and complexity grow towards the one million equation threshold, and beyond.

For example, the French Electrical Transmission System Operator RTE decided several years ago to use Modelica to model and simulate national and continental-wide power transmission systems. However, limitations in existing Modelica technology were such that they could use Modelica to generate the code of individual components, but then had to write their own simulation engine in the Dynaωo software (Masoom et al. 2021) to build and simulate the systems of their interest at the required scale, within the time frames allotted for real-time monitoring and management of the French power grid.

To overcome these inefficiencies, four years ago some of the authors of this paper started a research line with three main goals:

1. Compile Modelica code into the simulation code with a low runtime footprint in terms of both memory and execution time, running on a range of different machines, from the workstations typically used by engineers to run their simulations, to embedded devices where Modelica models can be deployed as part of control systems.
2. Exploit arrays of variables, equations, and models as first-class citizens to drastically cut code generation time and generated code size and improve simulation run time. (Schuchart et al. 2015) (Otter and Elmqvist 2017)

3. Skip the traditional C-code generation step in favour of generating LLVM-IR code that could be directly turned into highly optimised machine code.

A very preliminary experimental development was reported in (Agosta et al. 2019), together with a first tentative roadmap.

Based on that first experience, which could be classified as TRL-2 and whose results seemed promising, the development of the MARCO (Modelica Advanced Research COmpiler) compiler was started and has since then grown into a full-fledged research project involving several master's and PhD students, as well as more senior faculty with Computer Science and with Automation expertise.

The purpose of this paper is thus to present to the Modelica community the current state of the art of this project, which has now reached TRL-4, together with an updated roadmap for future work. Some interesting results obtained on non-trivial case studies will also be reported.

The paper is structured as follows: in Section 2, the MARCO compiler's objectives, requirements, and specifications are stated. Section 3 briefly describes the compiler architecture and its current development status. Section 4.3 reports the results of two non-trivial case studies inspired by real-life applications. Section 6 concludes the paper with some final remarks.

2 Goals, Requirements, Specifications

The main goal of MARCO is to experiment with methodologies and algorithms to generate the fastest possible executable code from large Modelica models, and to do it quickly and efficiently. The medium-term objective is to handle models with one million to ten million differential-algebraic equations (DAEs), eventually reaching 100 million in the long term, although this may require more fundamental breakthroughs.

At the time of writing, MARCO is not primarily meant to be a production-grade compiler. As such, it does not aim at covering the complete range of models that can be written in Modelica. The idea is to *demonstrate* the capability of generating fast executable code fast on a subset of large-scale system models that can be written in the Modelica language, that are however relevant for industrial application domains. This sub-set could then be progressively enlarged as time passes, possibly – but not necessarily – covering the full range of models that can be written using the Modelica language. At some point, MARCO could turn into industrial-grade software, or alternatively be used as a research prototype for implementing such software; it is currently too early to say that.

This project is specifically interested in monitoring and improving metrics related to the quality of the tool, on top of the quality of the result. Thus, MARCO aims at optimising the time-to-solution, which is to be intended as the sum of the time required to generate an executable simulation, plus the proper simulation time of a Modelica model.

Efficient handling of arrays of models and equations is an essential feature to fulfil this goal. Arrays should be handled as first-class citizens throughout the entire toolchain, avoiding expanding them into their scalar constituents, unless strictly required, thus shortening the structural analysis time and the executable code generation time.

To generate efficient runtime simulation code, the mathematical structure of the problem should be preserved as much as possible throughout the toolchain, and exploited during its latest stages to allow the generation of more efficient machine code.

One important point when handling very large systems with over a million variables and equations concerns handling the simulation results. The default behaviour of Modelica compilers is to save all the variable values at each reporting time step, possibly skipping protected components. However, for such large-sized models, this approach easily leads to massive, multi-GB-sized simulation results files, which are unnecessarily cumbersome and largely useless since most of those variables have little or no specific interest for end-users that generally on a relatively small subset of relevant output variables.

The idea is then not only to avoid wasting CPU time and disk space to store all the simulation results but actually to structure efficient simulation code around the fact that only some variables (which can be declared, e.g., as top-level outputs or listed in a custom annotation) are interesting for the end user. For example, if a certain variable is only of interest as an intermediate computation step towards the computation of the state derivatives, the generated code could only store it in some CPU registers so that not only the time to save it to disk is saved, but also the time to shuffle it back and forth from the CPU cache to RAM. In some cases, the computation of certain variables could even be skipped outright.

Along the same line, Modelica compilers usually generate code that allows changing parameter values at runtime without re-building the simulation executable from scratch. This is of course essential if the build time is comparable or even larger than the simulation run time, as it often is with current Modelica tools. However, this has a price in terms of less efficient simulation code because of additional indirection and memory access, as well as leaving less room for extreme machine-code optimisations.

Also, during typical simulation-based studies (including parameter optimisation), only a relatively small number of parameters are subject to change; these are a tiny fraction of the complete set of parameters for million-equations models, which could count tens of thousands or more parameters. Since the goal of MARCO is to generate code which is as fast as possible and to do it as fast as possible, all parameters that are determined by binding equations should be constant-evaluated at compile time. Although it should remain possible to make some exceptions to support parameter-sweeping or parameter-optimisation studies, the (few) parameters *not* to be constant-evaluated

DOI
10.3384/ecp20413

should be declared explicitly.

3 Current Status of MARCO

In this section, we provide an overview of the current status of MARCO with respect to the objectives we are aiming to achieve. First, we focus on the software architecture chosen, which reflects the state-of-the-art in compiler design and construction. Then we follow with a discussion of the features currently supported by the compiler, both in terms of Modelica language features, and in terms of the set of runtime solvers currently supported for the simulation code generation.

3.1 Software Architecture

MARCO is written using the C++ language, and it is based on the technologies developed within the LLVM project (Lattner and Adve 2004). The LLVM project is a collection of modular and reusable compiler technologies. Its most important part are the LLVM core libraries (often simply referred as LLVM), which provide a modern target-independent optimizer and code generator for an increasing amount of processor architectures. LLVM is a mature project heavily used both in the industry and in compiler research, and in MARCO we rely on it to implement the back-end of the compiler, which therefore outputs machine code directly instead of C code. Is is worth noting that using LLVM and its intermediate representation (LLVM-IR) enables the reuse of the backend optimisations provided by LLVM itself and the possibility of targeting different architectures – i.e. ARM-based embedded systems and not just PCs based on Intel processors – without the need to implement any additional transformations.

The front-end of MARCO is based on MLIR, also part of the LLVM project, which represents a novel approach to building reusable and extensible compiler infrastructures (Lattner, Amini, et al. 2021). Before MLIR, compiler front-ends were often built from the ground up, because different language features call for different internal data structures for the code – or, in other words, different intermediate representations. However, while these intermediate representations may differ from each other, there is a set of common abstract tasks performed on such representations that does not depend on the semantic of each operation in the code. MLIR provides a construction set, so-to-speak, where the compiler developer only has to declaratively define the set of domain-specific intermediate representations they need – called *dialects* – based on simple concepts like *operations*, *types*, and *attributes*. Additionally, MLIR provides built-in dialects for semantics that history has shown to be common amongst multiple programming languages. The implementation of new dialects and the combination of existing ones contribute to the definition of Multiple Layers of Intermediate Representations of the code, from which the MLIR acronym stems. In summary, MLIR allows to build compilers with less human effort, providing a library of primitives that previously needed to be rewritten from scratch for each different language.

From a high-level point of view, MARCO is composed of multiple libraries organized as a pipeline. This pipeline is overall similar to the familiar one known from the literature (Cellier and Kofman 2006) and already employed in other state-of-the-art Modelica compilers like the OpenModelica Compiler. In our compiler design, however, all the steps required for the causalization of the model are performed through successive transformations of a new MLIR dialect explicitly devised for the Modelica language. In addition, at the end of the pipeline, the causalized model in MLIR dialect form is translated into LLVM-IR code, the intermediate representation (IR) used by LLVM. Then, we exploit LLVM to translate such code into an object file, which is then linked with the MARCO runtime library to obtain the executable simulation. The translation to LLVM-IR exploits the existing MLIR built-in dialects and transformations to the maximum possible extent, greatly reducing the workload required for its implementation.

The MARCO runtime library is also written in C++, and serves two purposes: the first is to provide the implementations for functions that are inconvenient to be represented directly using LLVM-IR; the second is to actually drive the simulation process, by leveraging other functions which are instead emitted during the compilation process, which typically provide information about the compiled model. It is worth noticing how, even if not strictly necessary for the generation of the simulation, the MARCO runtime library enables faster development and testing, together with the possibility of using more complex solutions – like multithreading – that would otherwise be way more difficult to handle.

3.2 Arrays & Flattening

The first step of the process that transforms a Modelica model into executable simulation code (Fritzson 2014) is the so-called *flattening* (Modelica Association 2021). During flattening, the models with their variables, parameters, and equations are instantiated according to the rules that govern name lookup, inheritance, and modular composition of Modelica models. This first step results in a set of variable and parameter declarations, a set of record type definitions, and a set of hybrid DAEs.

The fundamental requirement for preserving arrays as first-class citizens is to carry out the flattening without expanding array variables into their scalar constituents and without unrolling array or for-loop equations into their scalar requirements.

Given the complexity of the Modelica language, this first step is rather involved and would require substantial development effort. Luckily, recent advances in the development of the OpenModelica Compiler (OMC), namely the new OMC front-end (Pop et al. 2019) provide this functionality out of the box. The new OMC front-end provides more than adequate performance also for very large models, as long as they are built by instantiating large ar-

rays of a comparably limited number of classes, which is typically the case in many systems-of-systems and smart grid applications. In this case, most of the flattening effort can be performed once for an array of components that may count hundreds or thousands of elements, thus slashing the flattening time dramatically.

Additionally, the new OMC front-end can also process models that contain a large number of individual instances of the same class with the same modifier structure, automatically collecting them in arrays before proceeding with the rest of the flattening process. This allows to process models of systems such as power grids (Bartolini, Casella, and Guironnet 2019), gas networks (De Pascali et al. 2022), or district heating networks (Long et al. 2021), that can be built automatically by translating graph-based system descriptions into Modelica system models, eventually transforming them into array-based models.

Recent advances in the definition of Base Modelica, formerly known as Flat Modelica (*MCP-0031: Base Modelica and MLS modularization* 2023), were used to interface the OMC new frontend and the MARCO compiler, with some extensions to support array-preserving model descriptions. Specifically, declarations of arrays of models are turned into the corresponding declarations of arrays of variables, where the variables of the model array become array variables; accordingly, the equations of the model array become array equations, declared via for-loops covering the entire array index range, see (Casella 2023) for some concrete examples.

MARCO thus accepts array-preserving Base Modelica textual models as inputs. Using a textual interface may be somewhat less efficient than directly accessing the OMC frontend internal data structures. On the other hand, relying on a high-level, reasonably stable, human-readable interface, which is presumably going to become a Modelica Association standard eventually, seems to be the best option for long-term development, without running the risk of relying on low-level on features that may change or become obsolete in the future.

This decision allowed to focus the development of MARCO on the current bottlenecks of the typical Modelica simulation workflow for very large models, namely the structural analysis, the code generation, and the runtime execution.

3.3 Supported Modelica Features

Therefore, MARCO relies on the OMC's new front-end for flattening, which is a complete, efficient implementation of the Modelica Language Specification, fully supporting the Modelica Standard Library. From this point of view, MARCO can accept models built with the most sophisticated features of Modelica, such as replaceable classes, conditional components, overconstrained connectors, stream variables, etc., which will be converted into flat hybrid DAE systems, possibly involving multi-dimensional arrays of variables.

Current limitations in the range of the models that MARCO can turn into efficient simulation code thus only regard the *mathematical* structure of the model rather than its *object-oriented* structure.

MARCO only supports continuous-time variables and equations at the time of this writing. Although we acknowledge that this limitation is particularly severe for practical applications, on the other hand, MARCO already enables us to demonstrate the scaling capabilities of the compiler with respect to the model's size. Support of discrete variables, event-handling, if-equations and when-equations is planned for the near future.

Thanks to recently developed array-based extensions of matching and sorting algorithms (Fioravanti et al. 2023), MARCO can very efficiently handle the causalization of array-based DAEs, including multi-dimensional arrays. The output of this phase is the matching of continuous slices of these arrays with corresponding for-loop equations and the ordering of their solution in Block Lower Triangular (BLT) form.

In fact, one interesting result proven in (Fioravanti et al. 2023) is that the optimal matching problem in the case of arrays (where optimal means that the arrays slices and corresponding array-equations should have the maximum possible size) is in general an NP-complete problem. Heuristics were then developed to efficiently handle roto-translation of index – i.e., cases where the equations in for-loop involve exchanging indexes and adding fixed offsets, as shown in Listing 1.

Listing 1. Example of array equations with fixed offset

```
  Real x[N, M];
  Real y{N, M];
equation
  for i in 2:N-1 loop
    for j in 1:M loop
      x[i,j] = y[j, i + 1] + x[i - 1, j];
    end for;
  end for;
```

MARCO supports arbitrary Modelica functions, possibly with inlining, with the exception of external functions, whose support is planned for the future. It can also differentiate functions using AD techniques (Neidinger 2010) whenever needed for Jacobian computations.

The support for records is currently being implemented, including the support of operator records, which is necessary for power system models using Complex numbers, a potentially very interesting application, in the near future.

At the time of writing, index reduction, dummy derivatives and state variable changes are not yet supported. Although this lack also represents a severe limitation for a Modelica compiler, there are some interesting application fields – e.g., modelling the thermal dynamics of buildings and district heating systems, as well as electro-mechanical modelling of power transmission and distribution systems – where these features are not needed.

3.4 Runtime Solvers

Regarding the runtime solvers, the initial goal of MARCO is to demonstrate its potential in two categories of application scenarios.

The first one involves non-stiff models that may be simulated with explicit ODE integration methods. This is also useful for co-simulation or real-time simulation, possibly running on embedded hardware using FMI or e-FMI. In this case, fixed-time-step explicit Euler's method is used. More sophisticated higher-order explicit integration methods such as Runge-Kutta could be implemented, but they do not represent a priority as long as MARCO remains a technology demonstrator rather than a full-fledged production-level tool.

In this context, it may be necessary to solve algebraic loops at each time step corresponding to strong components in the BLT. Currently, MARCO is restricted to small linear systems that can be solved efficiently in closed-form by symbolic manipulation. The integration of sparse linear (KLU) and nonlinear (Kinsol) solvers with symbolic Jacobian code generation is planned for the near future.

The second scenario instead involves systems which are stiff or involve large algebraic systems of equations. In this case, the design choice was to rely on the open-source DAE solver IDA from the Sundials tool suite (Gardner et al. 2022), which provides the efficient solution of large, sparse DAE models using BDF algorithms, with adaptive step size and error control.

IDA optionally requires the (sparse) Jacobians of the DAE formulation of the system with respect to all the variables and to the state derivatives to solve the implicit BDF formula, and it goes without saying that an overall efficient implementation requires computing such Jacobians analytically, to reduce the time spent computing Jacobians and also to avoid unnecessary iterations of the BDF solver caused by poor Jacobians. MARCO is thus endowed with automatic differentiation algorithms and generates efficient code to compute the Jacobians required by IDA.

To reduce the size of the implicit system that IDA needs to solve at each iteration of the solution of the BDF formula, the results of the causalization algorithm are exploited: instead of passing to the IDA solver the complete DAE system $F(x, \dot{x}, v, t) = 0$), where x is the vector of state variables, v the vector of all algebraic variables, and t the time variable, a reduced system of equations $F_r(x, \dot{x}, w, t)$ is passed instead, where w is the vector of the algebraic variables that are unknowns of implicit systems; the other algebraic variables are computed by sequences of assignments that correspond to the explicit solutions of equations that have 1×1 blocks on the BLT diagonal (Scuttari et al. 2023). In other words, IDA is directly used to solve the linear and nonlinear implicit algebraic equations and the (stiff) differential equations, while the results of the causalization steps are used to compute all the other variables explicitly in the generated code.

While a single thread currently carries out the sequential part of the residual computation, the subsequent computation of the residuals matched to $\dot{x}$ and w and of the Jacobian element is carried out by parallel threads, since they can be computed independently. In the future, also the sequential part could be parallelised.

Initial equations are also solved using IDA, which acts as an interface to the underlying sparse Kinsol solver. Currently, MARCO can only handle square non-singular initialisation problems, where the number of initial equations matches the number of differentiated variables plus the number of `fixed = false` parameters. The solution of under and over-determined initialisation problems, which is closely related to index reduction and dummy derivatives, is currently not yet supported.

Last but not least, MARCO only outputs to the CSV result file the top-level output variables of the model. An extension of the array-aware matching and sorting algorithm along the lines of (Manzoni and Casella 2011) could identify the system equations and variables that are strictly needed to compute the state derivatives and the top-level system outputs, allowing to skip the computation of all other algebraic variables defined in the model. This feature, which is planned for the near future, will further optimise the simulation time.

4 Case Studies

In this section, the results of two case studies are reported to demonstrate the current capabilities of the MARCO compiler. These case studies are motivated by real-life applications; they are simple enough to be contained in a few dozen lines of code (see the Appendix) but are nevertheless definitely non-trivial to handle, in particular as regards the need for matching slices of the arrays to subsets of for-loop equations involving them. Also, both case studies are easily scalable via parameters to test the tool's performance with the increasing model size.

4.1 3D Thermal Model of a Microchip

Modern microprocessors feature higher and higher power density, requiring more and more advanced fluid-based cooling systems. These models combine 0D and 1D cooling system models, which are conveniently represented in Modelica, with 3D thermal models of the microchip body and heat sink body, which need high spatial definition to identify potentially harmful hot spots.

This is currently achieved by co-simulation set-ups (Terraneo et al. 2022), where the microchip thermal dynamics are simulated by a separate dedicated simulation tool. However, it would be quite convenient to embed a detailed 3D thermal model of the microchip directly within the Modelica model, avoiding the complication and inconvenience of the co-simulation setup.

This first case study thus demonstrates the capability of MARCO to handle high spatial resolution 3D thermal models of solid bodies. The 3D thermal model is built in a fully object-oriented way, by first defining an elementary 0D Volume model, with a lumped thermal capacitance in

the middle and 6 thermal conductances and 6 thermal ports in the directions east, west, north, south, up, and down. The microchip thermal model is assembled by instantiating a 3D $N \times M \times P$ array of such 0D models and connecting them via for loops.

This approach leads to a very compact Modelica source code with just three for loops, one for each orthogonal spatial direction. The alternative would be to write the discretised 3D heat equations directly in the body of the model, but that requires handling the inner volumes, the face volumes, the edge volumes, and the corner volumes differently, leading to a much longer and error-prone code with many more for loops, and a much higher likelihood of making some mistakes with the loop indices when writing the equations for the various cases.

Furthermore, with the full object-oriented approach, one can connect heat sources or other thermal objects of arbitrary shape to *portions* of the outer faces of the 3D microchip model, e.g. representing specific active semiconductor areas on the microchip surface; unconnected outer faces are automatically considered as thermally insulated, thanks to the default connection equations `Q_flow = 0` generated by the compiler for unconnected thermal ports. Handling the non-trivial geometry of such active areas without using connection equations becomes extremely complicated and counter-intuitive, as one would also need to write specific for-loop equations for each rectangular *thermally insulated* region.

In the present case study, for simplicity, half of the lower face of the chip (corresponding to the active semiconductor area) was connected to a uniformly distributed 2D heat source, while the other half was left unconnected, and thus insulated. More elaborate setups could be conceived, e.g. representing active cores on the chip surface.

The upper face of the microchip is instead connected to a 2D fixed temperature source, corresponding to the surface of the heat sink block. A more realistic model could include a full thermal model of the heat sink and its cooling system.

This object-oriented model contains a huge number $N_c = O(NMP)$ of connection equations; each face-to-face connection of two adjacent 0D blocks generates a small system of linear equations, corresponding to the series connection of the two half-conductances of the adjacent 0D blocks in that direction. However, these systems can be easily solved in closed form, corresponding to the well-known formula for the conductance of series-connected conductors, allowing to explicitly compute the heat flow between the capacitances of adjacent 0D blocks in each spatial direction *without even computing the temperature at the block boundaries*. Additionally, thanks to the array-preserving nature of MARCO, this symbolic solution needs to be carried out only once during code generation, so it takes a negligible amount of time.

For the sake of this simple case study, only eight output temperatures were computed and saved, namely the temperatures at the four corners of the upper and lower faces of the microchip. This enabled the comparison of the simulation results with those obtained with OpenModelica.

The thermal microchip model was simulated both by explicit Euler's algorithm and by IDA, in a test case of increasing size, up to $M = N = 96$, $P = 32$, which leads to a model with over 4 million DAEs and about 250,000 state variables. The simulation starts at thermal equilibrium with zero thermal power input and simulates the response of the system to a step increase of the thermal flux applied to half of the bottom face of the microchip.

4.2 Heat Exchanger Network with Methanol

Another interesting field of application that can easily lead to large-scale models is thermo-fluid systems, as found in large industrial plants, district heating systems, and smart distributed energy systems in general. When modelling such systems, non-trivial fluid property models are often needed and computed using functions.

The second test case tries to capture the main features of these applications in a simple and scalable test model. The model contains a 2D $N_u \times N_h$ array of heat exchangers, which are arranged in N_u sequential rows, each containing N_h parallel heat exchangers, whose outlet flows are then mixed before being distributed to the next row. Each heat exchanger is then modelled with N_v finite volumes. The mass flow rates and heat flows of each heat exchanger are time-varying, and set up in a way that guarantees that no two heat exchangers ever operate at the same temperature. Overall, the number of variables and DAEs of the system model is $O(N_u N_h N_v)$.

The compressibility of the fluid inside the heat exchangers is neglected for simplicity, leading to trivial mass balance equations. On the other hand, an accurate model of the relationship between the temperature and the specific energy and enthalpy was developed using Modelica functions, using results from (Craven and de Reuck 1986).

4.3 Experimental Results

The results of the simulations of medium-size models were successfully compared with the simulation results obtained with the OpenModelica tool, using the same solution algorithm, and were found to agree with the results produced by MARCO with high accuracy. Then, MARCO was used to simulate the much larger instances of the test cases mentioned in the previous two sub-sections, which are beyond the current capabilities of the OpenModelica compiler.

All tests were conducted on a server with an i9-12900KF Intel processor and 96 GB of RAM, running Linux Ubuntu 20.04 LTS. At the time of this writing, the results obtained with IDA, although correct, are still not efficient as expected, so the results summarised in Table 1 and shown in Figures 1 and 2 only refer to explicit Euler's method. Results with IDA are expected to be available for the final revision of this paper.

A direct comparison of the performance of MARCO against other Modelica tools is beyond the scope of this

DOI
10.3384/ecp20413

Table 1. Compilation and simulation times for largest simulated models.

	Equations	Steps	Compilation time [s]	Simulation time [s]
Heat exchangers network	14776202	40000	10.13	33574.76
3D thermal chip	4465160	60000	86.24	605.47

paper. It is worth mentioning, though, that the authors are not aware of any other Modelica tool which is currently able to handle models with 15 million equations or at least to do so with compile times of a few tens of seconds on low-cost hardware (a 1,500 € gaming machine).

5 Roadmap

The results presented in the previous section demonstrate that the MARCO compiler can handle non-trivial, array-based, very large models with very short compile times and good runtime performance, on a scale of model sizes currently inaccessible to mainstream Modelica tools.

On the other hand, the class of models that can be currently handled is minimal. Future development work is thus planned in different directions.

Record and operator record handling is currently being addressed and could be completed in time for the final version of this paper. Combined with the support of hybrid systems, this could make MARCO capable of handling large-scale power system models (Bartolini, Casella, and Guironnet 2019), potentially allowing it to replace the parts of Dynaωo (Masoom et al. 2021) that currently take care of assembling the whole system model starting from the generated C code of individual components.

The addition of external function handling could also allow tackling models of advanced microchip cooling systems, including detailed 3D thermal dynamics end using refrigerant models from the ExternalMedia library (Casella and Richter 2008).

FMI export and embedded code generation are other promising areas of development.

Finally, handling index reduction, dummy derivatives, and under/overconstrained initialisation problems could prove to be very hard. One option for index reduction, as already planned in (Agosta et al. 2019), is to solve these problems on a fully scalarised set of equations, adding the extra differentiated scalar equations to the system that are needed to make it index-1. All other equations would still be handled in an array-preserving way, still leading to a substantial performance advantage compared to the traditional flat-scalar equation tools.

In the long term, we plan to leverage support from both the Modelica and LLVM communities. To this end, we plan to release MARCO as an open-source project once record handling and hybrid system support are available, providing the capability to address a sufficiently large number of real-world large-scale problems.

6 Conclusions

This paper introduced the MARCO compiler, which is currently developed at the Dipartimento di Elettronica, Informazione e Bioingegneria of Politecnico di Milano in collaboration with Edinburgh Napier University. MARCO aims at demonstrating algorithms and methodologies to compile large-scale Modelica models efficiently, producing fast binary code. It accepts flat, array-preserving Base Modelica code as input, produced by the new front-end of the OpenModelica compiler, and causalizes it using novel array-preserving matching and sorting algorithms. Executable code is generated using the LLVM framework:

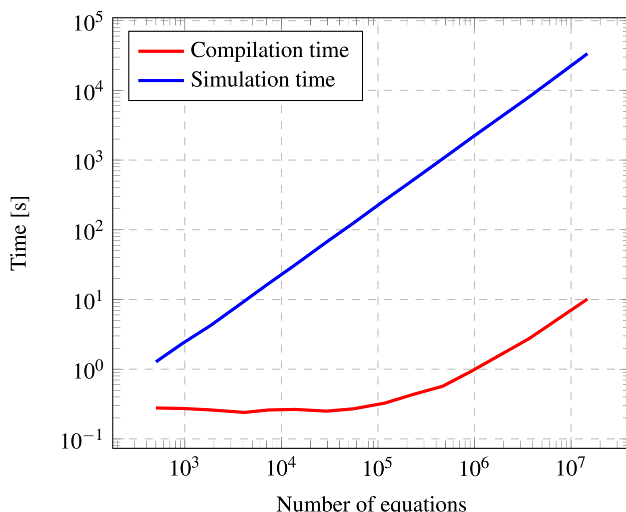

Figure 1. Heat exchangers network model

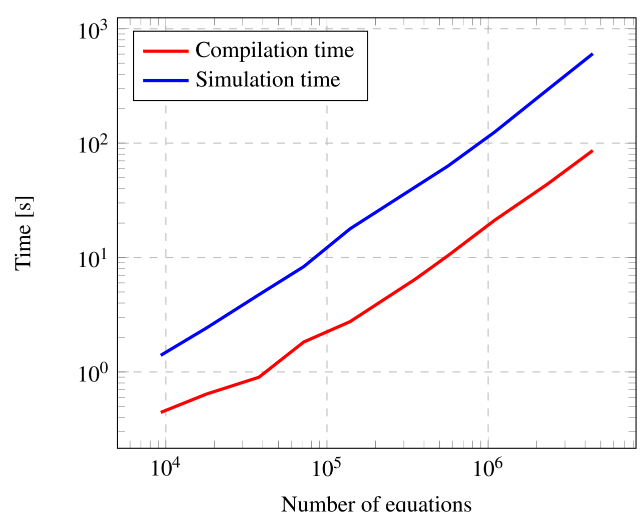

Figure 2. 3D thermal chip model

an LLVM-IR is first produced by the compiler and then directly turned into efficient, architecture-optimised executable code.

Thanks to the complete support of the Modelica language provided by the OpenModelica front-end, MARCO is capable of handling Modelica code using all the advanced object-oriented features of the language. Its current limitations regard the mathematical structure of the flat system, which currently needs to be an index-1, purely continuous-time dynamical system, possibly involving Modelica functions.

The generated code can simulate dynamical systems using explicit Euler's algorithm or by using the IDA DAE solver, in which case the code to compute symbolic Jacobians is also generated.

The current capabilities of the MARCO compiler were demonstrated on two large-scale test cases: 3D object-oriented thermal models of a microchip with up to 4 million equations, and equation-based models of networks of heat exchangers with a detailed function-based fluid model, with up to 15 million equations. In both cases, the compilation time is at most a few tens of seconds and is one or more orders of magnitude less than the run time. To the authors' knowledge, no other Modelica tool can handle Modelica models at this scale.

Future developments of MARCO in the short term regard the implementation of operator records and event handling, at which point the release of MARCO as open-source software is planned. These two additional features will enable the compilation and simulation of national- and continental-scale power system models such as those of the ScalableTestGrids library (Bartolini, Casella, and Guironnet 2019).

Medium- and long-term developments include supporting external functions, code generation for embedded hardware, FMI export, index reduction, and under/over-constrained initialisation problems.

Acknowledgements

The authors are grateful to former MSc students Massimo Fioravanti and Nicola Camillucci, who contributed to the development of the MARCO code base during the last three years.

References

Agosta, Giovanni et al. (2019-03). "Towards a High Performance Modelica Compilers". In: *Proc. 13th International Modelica Conference*. Ed. by Anton Haumer. Regensburg, Germany, pp. 313–320. DOI: 10.3384/ecp19157313.

Bartolini, Andrea, Francesco Casella, and Adrien Guironnet (2019-03). "Towards Pan-European Power Grid Modelling in Modelica: Design Principles and a Prototype for a Reference Power System Library". In: *Proc. 13th International Modelica Conference*. Ed. by Anton Haumer. Regensburg, Germany, pp. 627–636. DOI: 10.3384/ecp19157627.

Casella, Francesco (2015-09). "Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives". In: *Proceedings 11th International Modelica Conference*. Ed. by Peter Fritzson and Hilding Elmqvist. The Modelica Association. Versailles, France, pp. 459–468. ISBN: 978-91-7685-955-1. DOI: 10.3384/ecp15118459.

Casella, Francesco (2023). *Simulation of large-scale Modelica models with array-preserving technology: early results and perspectives*. URL: https://tinyurl.com/OMWorkshopKeynote2023 (visited on 2023-05-24).

Casella, Francesco and Christoph C. Richter (2008-03). "ExternalMedia: a Library for Easy Re-Use of External Fluid Property Code in Modelica". In: *Proceedings 6th International Modelica Conference*. Ed. by Bernhard Bachmann. Modelica Association. Bielefeld, Germany, pp. 157–161. URL: http://www.modelica.org/events/modelica2008/Proceedings/sessions/session2b1.pdf.

Cellier, F. E. and E. Kofman (2006). *Continuous System Simulation*. Springer-Verlag.

Craven, R. J. B. and K. M. de Reuck (1986). "Ideal-Gas and Saturation Properties of Methanol". In: *International Journal of Thermophysics* 7.3, pp. 541–552.

De Pascali, Matteo Luigi et al. (2022-06). "Flexible object-oriented modelling for the control of large gas networks." In: *Proc. 11th IFAC Symposium on Control of Power and Energy Systems, IFAC PapersOnLine*. Vol. 55. 9. Online, pp. 315–320. DOI: 10.1016/j.ifacol.2022.07.055.

Fioravanti, Massimo et al. (2023-07). "Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models". In: *ACM Trans. Math. Softw.* Just Accepted. ISSN: 0098-3500. DOI: 10.1145/3611661. URL: https://doi.org/10.1145/3611661.

Fritzson, P. (2014). *Principles of Object Oriented Modeling and Simulation with Modelica 3.3*. Wiley IEEE Press.

Gardner, David J et al. (2022). "Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)*. DOI: 10.1145/3539801.

Lattner, Chris and Vikram Adve (2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proc. Int'l Symp. on Code Generation and Optimization*. Palo Alto, California. ISBN: 0-7695-2102-9.

Lattner, Chris, Mehdi Amini, et al. (2021). "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

Long, Nicholas et al. (2021-09). "Modeling district heating and cooling systems with URBANopt, GeoJSON to Modelica Translator, and the Modelica Buildings Library," in: *Proceedings of the 17th IBPSA Conference*. Bruges, Belgium, pp. 2187–2194. DOI: 10.26868/25222708.2021.30943.

Manzoni, Vincenzo and Francesco Casella (2011-03). "Minimal Equation Sets for Output Computation in Object-Oriented Models". In: *Proceedings 8th International Modelica Conference*. Ed. by C. Clauss. Modelica Association. Dresden, Germany, pp. 784–790. ISBN: 978-91-7393-096-3. DOI: 10.3384/ecp11063784. URL: http://www.ep.liu.se/ecp/063/088/ecp11063088.pdf.

Masoom, Alireza et al. (2021-08). "Modelica-based simulation of electromagnetic transients using Dynaωo: Current status and perspectives". In: *Electric Power Systems Research* 197, p. 107340. DOI: 10.1016/j.epsr.2021.107340.

MCP-0031: Base Modelica and MLS modularization (2023). URL: https://github.com/modelica/ModelicaSpecification/tree/MCP/0031/RationaleMCP/0031 (visited on 2023-05-24).

Modelica Association (2021-02). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.5*. Tech. rep. Linköping: Modelica Association. URL: https://specification.modelica.org/maint/3.5/MLS.html.

Neidinger, Richard D (2010). "Introduction to automatic differentiation and MATLAB object-oriented programming". In: *SIAM review* 52.3, pp. 545–563.

Otter, Martin and Hilding Elmqvist (2017). "Transformation of Differential Algebraic Array Equations to Index One Form". In: *Proceedings of the 12th International Modelica Conference*. Linköping Electronic Conference Proceedings. Linköping University Electronic Press. URL: https://elib.dlr.de/117431/.

Pop, Adrian et al. (2019-03). "A New OpenModelica Compiler High Performance Frontend". In: *Proc. 13th International Modelica Conference*. Ed. by Anton Haumer. Regensburg, Germany, pp. 689–698. DOI: 10.3384/ecp19157689.

Schuchart, Joseph et al. (2015). "Exploiting repeated structures and vectorization in modelica". In: *Proc. of the 11th Int. Modelica Conference, Versailles. www. ep. liu. se/ecp/118/028/ecp15118265. pdf*.

Scuttari, Michele et al. (2023). "Clever DAE: Compiler Optimizations for Digital Twins at Scale". In: *2nd Annual Compiler Frontiers Workshop*. ACM Press. DOI: 10.1145/3587135.3589945.

Terraneo, Federico et al. (2022). "3D-ICE 3.0: Efficient Nonlinear MPSoC Thermal Simulation With Pluggable Heat Sink Models". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.4, pp. 1062–1075. DOI: 10.1109/TCAD.2021.3074613.

TestCases (2023). URL: https://casella.faculty.polimi.it/transfer/MARCO_Tests.zip (visited on 2023-05-24).

Appendix: Source code of the test cases

The code of the two test cases is reported in this Appendix for the reader's convenience. The code has been edited for conciseness, the full packages can be downloaded from (*TestCases* 2023).

Listing 2. Source code of the 3D thermal model of a microchip

```
package ThermalChipOO
 package Interfaces
  connector HeatPort
   Real T;
   flow Real Q;
  end HeatPort;
 end Interfaces;

 package Models
  model Volume
   parameter Real lambda = 148;
   parameter Real rho = 2329;
   parameter Real c = 700;
   parameter Real Tstart = 273.15 + 40;
   parameter Real C, Gx, Gy, Gz;
   Interfaces.HeatPort upper, lower;
   Interfaces.HeatPort left, right;
   Interfaces.HeatPort top, bottom;
   Interfaces.HeatPort center;
   Real T(start = Tstart, fixed = true);
  equation
   C*der(T) = upper.Q + lower.Q +
              left.Q + right.Q +
              top.Q + bottom.Q + center.Q;
   upper.Q  = Gx*(upper.T  - T);
   lower.Q  = Gx*(lower.T  - T);
   left.Q   = Gy*(left.T   - T);
   right.Q  = Gy*(right.T  - T);
   top.Q    = Gz*(top.T    - T);
   bottom.Q = Gz*(bottom.T - T);
   center.T = T;
  end Volume;

  model TemperatureSource
   Interfaces.HeatPort port;
   Real T = 298.15;
  equation
   port.T = T;
  end TemperatureSource;

  model PowerSource
   Interfaces.HeatPort port;
   input Real Q;
  equation
   port.Q = -Q;
  end PowerSource;

  partial model BaseThermalChip
   parameter Integer N;
   parameter Integer M;
   parameter Integer P;
   parameter Real L = 12e-3;
   parameter Real W = 12e-3;
   parameter Real H = 4e-3;
   parameter Real lambda = 148;
   parameter Real rho = 2329;
   parameter Real c = 700;
   parameter Real Tstart = 273.15 + 40;
   parameter Real l = L/N;
   parameter Real w = W/M;
   parameter Real h = H/P;
   parameter Real Tt = 273.15 + 40;
   parameter Real C = rho*c*l*w*h;
   parameter Real Gx = lambda*w*h/l;
   parameter Real Gy = lambda*l*h/w;
   parameter Real Gz = lambda*l*w/h;

   Volume vol[N,M,P](
    each T(start = Tstart, fixed = true),
    each C = C, each Gx = 2*Gx,
    each Gy = 2*Gy, each Gz = 2*Gz);

   TemperatureSource Tsource[N,M]
    (each T = Tt);

   output Real Tct1 = vol[1,1,1].T;
   output Real Tct2 = vol[1,N,1].T;
   output Real Tct3 = vol[N,N,1].T;
   output Real Tct4 = vol[N,1,1].T;
   output Real Tcb1 = vol[1,1,P].T;
   output Real Tcb2 = vol[1,N,P].T;
   output Real Tcb3 = vol[N,N,P].T;
   output Real Tcb4 = vol[N,1,P].T;
  equation
   for i in 1:N loop
    for j in 1:M loop
     connect(vol[i,j,1].top,
             Tsource[i,j].port);
     for k in 1:P-1 loop
      connect(vol[i,j,k].bottom,
              vol[i,j,k+1].top);
     end for;
    end for;
   end for;
   for i in 1:N loop
    for k in 1:P loop
     for j in 1:M-1 loop
      connect(vol[i,j,k].right,
              vol[i,j+1,k].left);
     end for;
    end for;
   end for;
   for j in 1:M loop
    for k in 1:P loop
     for i in 1:N-1 loop
      connect(vol[i,j,k].lower,
              vol[i+1,j,k].upper);
     end for;
```

```
    end for;
   end for;
  end BaseThermalChip;

  model ThermalChipSimpleBoundary
   extends BaseThermalChip;
   parameter Real Ptot = 100;
   parameter Real Pv = Ptot/(N*M/2);
   PowerSource Qsource[N,div(M,2)]
    (each Q = Pv);
  equation
   connect(Qsource.port,
          vol[:,1:div(M,2),P].center);
  end ThermalChipSimpleBoundary;
 end Models;
end ThermalChipOO;
```

Listing 3. Source code of the heat exchanger network

```
package MethanolHeatExchangersDAE
 package Models
  model MethanolHeatExchangers
   parameter Integer Nu = 3;
   parameter Integer Nh = 4;
   parameter Integer Nv = 6;
   parameter Real w_nom = 1;
   parameter Real Q_nom = 500e3;
   parameter Real f_w = 1/30;
   parameter Real f_Q = 1/100;
   parameter Real T0 = 493.15;
   parameter Real V = 1;
   parameter Real beta = 0.01;
   parameter Real UA_nom = 10000;
   parameter Real alpha = 0.8;
   parameter Real Cw = 10000;
   parameter Real p_nom = 20e5;
   parameter Real V_v =
    V*(1-beta)/(Nu*Nh*Nv);
   parameter Real V_m = V*beta/Nu;
   parameter Real C_wv = Cw/(Nu*Nh*Nv);
   constant Real pi = 3.14159265359;
   Real w, w_h;
   Real Q[Nh], Q_c[Nu,Nh,Nv];
   Real T[Nu,Nh,Nv+1];
   Real h[Nu,Nh,Nv+1], h_m[Nu];
   Real T_tilde[Nu,Nh,Nv]
    (each start = T0, each fixed = true);
   Real T_w[Nu,Nh,Nv]
    (each start = T0, each fixed = true);
   output Real T_m[Nu]
    (each start = T0, each fixed = true);
   Real rho[Nu,Nh,Nv], rho_m[Nu];
   Real cv[Nu,Nh,Nv],  cv_m[Nu];
  equation
   w = w_nom*(1 + 0.2*sin(2*pi*f_w*time));
   w_h = w / Nh;
   for j in 1:Nh loop
    Q[j] = Q_nom/(Nu*Nh)*
      (1 + sin(2*pi*f_Q*time + 2*pi*j/Nh));
   end for;
   for j in 1:Nh loop
    T[1,j,1] = T0;
   end for;
   for i in 2:Nu loop
    for j in 1:Nh loop
     T[i,j,1] = T_m[i-1];
    end for;
   end for;
   T_tilde = T[:,:,2:Nv + 1];
   for i in 1:Nu loop
    V_m*rho_m[i]*cv_m[i]*der(T_m[i]) =
      w_h*sum(h[i,j,Nv+1] for j in 1:Nh) -
      w*h_m[i];
    for j in 1:Nh loop
     for k in 1:Nv loop
      (V_v*rho[i,j,k]*cv[i,j,k])*
          der(T_tilde[i,j,k]) =
        w_h*(h[i,j,k] - h[i,j,k+1]) +
          Q_c[i,j,k];
      C_wv*der(T_w[i, j, k]) =
        Q[j]/Nv - Q_c[i,j,k];
      Q_c[i,j,k] = UA_nom/(Nu*Nh*Nv)*
        (w/w_nom)^alpha*(T_w[i,j,k] - T_tilde[i,
            j,k]);
     end for;
    end for;
   end for;
   for i in 1:Nu loop
    rho_m[i] = p_nom/Methanol.R*T_m[i];
    h_m[i] = Methanol.h_T(T_m[i]);
    cv_m[i] = Methanol.cv_T(T_m[i]);
    for j in 1:Nh loop
     for k in 1:Nv loop
      rho[i,j,k] = p_nom /
        (Methanol.R*T_tilde[i,j,k]);
      cv[i,j,k] =
        Methanol.cv_T(T_tilde[i,j,k]);
     end for;
     for k in 1:Nv+1 loop
      h[i,j,k] = Methanol.h_T(T[i,j,k]);
     end for;
    end for;
   end for;
  end MethanolHeatExchangers;

  package Methanol
   constant Real R = 8.314462/32.04e-3;
   constant Real Tc = 512.64;
   constant Real f[8] =
    {3.90086, 10.9929, 18.3371, -16.3663,
     -6.22334, 2.80358, 1.07783, 0.96967};
   constant Real g[8] =
    {0.0, 4.12575, 3.26973, 3.77492,
     2.93574, 8.23747, 10.3312, 0.53326};

    function cp_T
     input Types.Temperature T;
     output Types.SpecificHeatCapacity cp;
    protected
     Types.PerUnit tau;
     Types.PerUnit u[8];
     Types.PerUnit x;
    algorithm
     tau := Tc / T;
     u := g * tau;
     x := f[1];
     for i in 2:8 loop
       x := x + f[i]*u[i]^2*exp(u[i])/
         (exp(u[i]) - 1)^2;
     end for;
     cp := x*R;
    end cp_T;

   function cv_T
    input Types.Temperature T;
    output Types.SpecificHeatCapacity cv;
   algorithm
    cv := cp_T(T) - R;
   end cv_T;

   function h_T
    input Types.Temperature T;
    output Types.SpecificEnthalpy h;
   protected
    Types.PerUnit tau;
    Types.PerUnit u[8];
    Types.PerUnit x;
   algorithm
    tau := Tc / T;
    u := g * tau;
    x := f[1]/tau;
    for i in 2:8 loop
     x := x + f[i]*g[i]/(exp(u[i]) - 1);
    end for;
    h := R*T*tau*x - 1361.810*tau/Tc;
   end h_T;
  end Methanol;
 end Models;
end MethanolHeatExchangersDAE;
```