Exploiting Repeated Structures and Vectorization in Modelica

Joseph Schuchart[1] Volker Waurich[2] Martin Flehmig[1]
Marcus Walther[1] Wolfgang E. Nagel[1] Ines Gubsch[2]

[1]Center for Information Services and High Performance Computing, TU Dresden, Germany
[2]Chair of Construction Machines and Conveying Technology, TU Dresden, Germany ,
{forename.surname}@tu-dresden.de

Abstract

Large and highly-detailed Modelica models are frequently modeled by utilizing repeated structures, which is a repetition of various elements that are linked together in an iterative manner. While the Modelica language standard supports the representation of repeated structures, most Modelica compilers do not exploit their advantages for efficient simulations. Instead, all repeated equations are flattened and all array variables are expanded. This leads to unnecessarily long compile times and higher memory consumption. Another aspect that has been yet inadequately considered and is closely connected to repeated structures is vectorization. The vector units of modern CPUs can be engaged to perform SIMD (Single Instruction, Multiple Data) operations, executing the same instruction on multiple data points in parallel. This reveals a high potential for faster simulations. This paper discusses the advantages of utilizing repeated structures for modeling in order to achieve both faster compilation and simulation times. The potentials of preserving for loops throughout compilation are demonstrated using a basic implementation in the OpenModelica Compiler. The effect on the simulation time by enabling vectorization is demonstrated for an appropriate model.
Keywords: SIMD, Vectorization, OpenModelica, Translation, Repetitions

1 Introduction and Related Work

The Modelica language is capable of describing large models with few lines of code by using repetitions of submodules. In general, submodels can be connected in an iterative manner by using `for` loops to express the repeated model structure. Models that contain repetitions are common in physical and technical modeling, e.g., models of battery packs and chain gears. Repeated structures can also be introduced by discretizing model elements, e.g., electrical wires or hydraulic pipes. Figure 1 represents a discretized model of an electrical wire. The underlying model is a distributed RC-model which

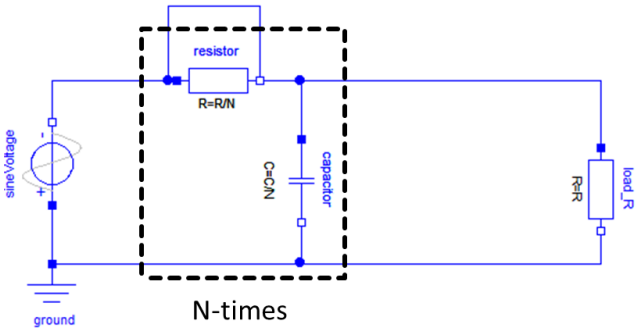


Figure 1. Distributed RC-model of an electrical wire using N repeated model elements.

is typically used to describe transmission behavior in electrical circuits. The circuit of resistor and capacitor is repeated N-times. Hence, all equations and variables which are defined in the resistor and capacitor model will appear N-times in the DAE system. Modelica compilers usually flatten the repeated equations without utilizing the information on repetitions. This can be explained by the fact, that the algorithms which transform the acausal DAE system into a causal, solvable ODE system have not been adopted to make use of repeated structures. Therefore, the symbolic manipulation handles an unnecessarily large amount of equations and variables and – as a consequence – the generated code does not contain any `for` loops anymore.

The potential of preserving repeated modules has already been illustrated (Zimmer, 2009). Zimmer describes the obvious potential of aggregating repeated modules and gives a basic method to approach the problem. He also highlights further issues regarding the symbolic algorithms in the compiler backend as will be explained later. In contrast to this work, the present paper will focus on iterated repetitions provided by `for` loops rather than by detecting modules. Another approach for addressing the topic has been proposed by Höger (Höger, 2011). By means of separate compilation of Modelica source code and subsequent linking, precompiled partial models can lead to smaller programs. Höger explains

the benefits of compiling source code before flattening the model but indicates problems related to the symbolic manipulation.

As has been stated already, symbolic manipulation for attaining a causal model needs further considerations. The main issues are related to index-reduction, typically performed by the Pantelides algorithm (Pantelides, 1988) and ordering of all equations and variables in a block-lower-triangular form, typically done by using the Tarjan algorithm (Tarjan, 1972). Arzt has provided a basic approach to adapt the Pantelides algorithm in order to exploit repeated structures (Arzt et al., 2014). Repeated equations and variables are collected in repetitive modules which share the same matching and derivation pattern. A prototypic implementation proved the ability to keep compilation time constant but has not yet been introduced into a usable Modelica compiler.

Preserving repeated structures throughout symbolic transformation can lead to decreased compilation times and memory consumption. It also reveals the potential for exploiting the vector units of a CPU efficiently by providing the underlying compiler with opportunities for creating vectorized code.

The remainder of the paper is structured as follows: In Section 2, a basic implementation for preserving repeated structures through the backend will be illustrated and the benefits of modeling `for` loops in causal sections are presented in Section 3. Section 4 describes the means and requirements to exploit vector computation and presents the impact on the simulation time of an example model. Finally, conclusions and an outlook are contained in Section 5.

2 Preserving Repeated Structures Throughout Compilation

Acausal Modelica models have to be transformed symbolically to attain a solvable, causal differential-algebraic system of equations (i.e., DAE system). Thus, every variable has to be assigned to a specific equation and a computation order has to be determined. The assignment between equations and variables is called matching and the ordering in a sequence of single equations and algebraic loops (i.e., systems of equations) is typically done by Tarjan's strongly connected component algorithm. If necessary, the index of the DAE system has to be reduced by performing the Pantelides algorithm. Besides these mandatory manipulations, there are even more Modelica-specific algorithms to improve the computable model, e.g., removal of trivial equations or tearing methods. These algorithms are well probed and standard in Modelica model compilation but only a few of them have been extended to handle repeated objects like the Modelica `for` loops. The mentioned algorithms can be applied easily to expanded `for` loops and array variables as it is the current default implementation in the OpenModelica Compiler.

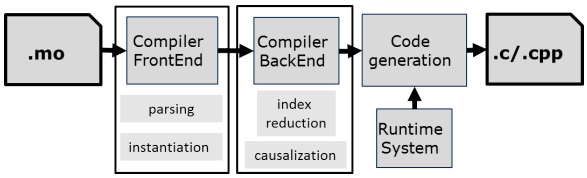


Figure 2. Compilation process in the OpenModelica Compiler.

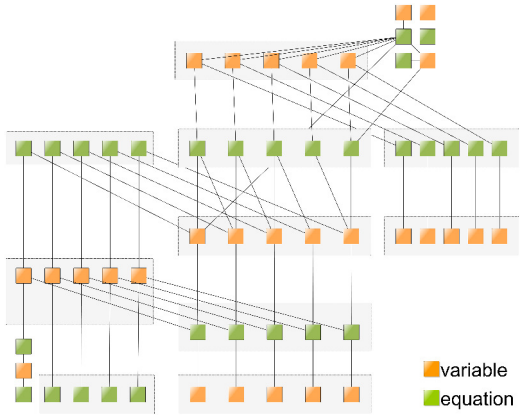


Figure 3. Flat model representation as a bipartite graph. Green squares represent equations and orange squares represent variables. Squares in a grey box can be gathered in a compact `for` loop notation or as an array-variable. Two vertices have a connecting edge if the variable appears in the equation. A removing of trivial equations has already been applied.

The workflow of the compilation process in the OpenModelica Compiler is depicted in Figure 2. After parsing the mo-file, all model elements are instantiated and a list of equations and variables, the flat DAE, is generated. This flat DAE representation is the basis for all further manipulations like index-reduction, causalization, and additional optimizations. The compiler backend creates a solvable model which is used to generate C or C++ code in order to compile an executable. If the `for` equations in the Modelica model should be used throughout the compilation procedure, the instantiation has to output a valid representation of the iterated equations for the repeated model elements. This is ongoing work within the frontend development and therefore is not covered by the presented paper.

In order to demonstrate the effects of retaining loops throughout compilation, a basic implementation in the compiler backend has been created. The instantiation still generates a flat DAE. However, the compiler backend now collects the flattened equations and array variables and establishes compacted `for` equations by comparing the terms of equations in order to find similarities. If a continuous iteration can be identified, an equivalent `for` equation is set up. For the wire model presented in Figure 1, the bipartite graph depicted in Figure 3 illustrates the compact notation of `for` equations.

DOI 10.3384/ecp15118265

As can be seen, the bipartite graph in Figure 3 has a repetitive structure. For this model, the number of repetitions is 5. The same structure can be expressed using the compact notation of `for` loops. There are two kinds of `for` loop equations that are passed through the compilation process. On the one hand, there is the repetition of several equations with differently indexed variables:

```
for i in 1:5 loop
  resistor[i].LossPower =
      resistor[i].v * resistor[i].i;
end for;
```

On the other hand, it is possible to have a single equation with a repetition of terms, e.g., in a summation that is represented by the `sum`-operator, for example:

```
ground.p.i + sineVoltage.p.i + load_R.n.i
  + sum((i->1:5) + capacitor[i].n.i) = 0.0
```

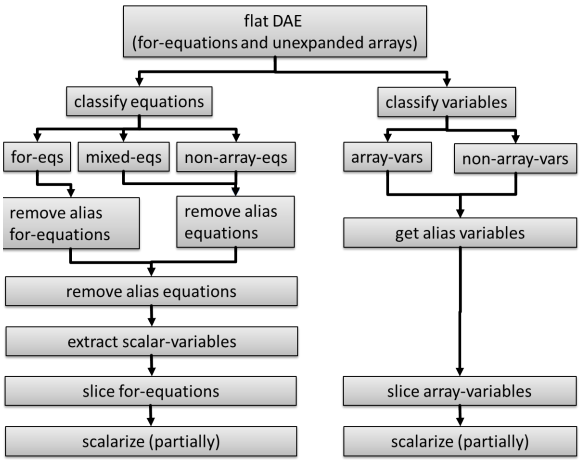


Figure 4. Overview of the process of collecting `for` equations.

2.1 Symbolic Transformation of Repeated Structures

The following section outlines the procedure of preserving `for` equations throughout model compilation. It basically covers the removal of trivial equations, matching, and identification of the computation sequence. The implemented prototype is only able to handle systems of maximum index one and without algebraic loops. Since this is a hard restriction, the fallback solution is to scalarize the DAE-system completely at any stage of compilation. At least the removal of trivial equations can benefit from the compact notation in that case. The results will include a benchmark for the removal of trivial equations exploiting `for` equations.

Since the information about repeated model elements is currently not available from instantiation, it has to be collected from the completely flattened DAE system.

To collect equations in `for` constructs, all flat equations have to be filtered for equations containing iterated variables. If an equation contains solely iterated variables, other equations which share the same algebraic terms without considering the array indexes are gathered as a `for` loop. Among these equations, the array indexes are compared and examined to determine the start and end index of the iteration. A linear iteration with a step of one is assumed. Equations containing several iterated instances of the same array variable are checked for possible terms like a summation of iterated variables. Since this process can be avoided by instantiating `for` equations and unexpanded arrays directly, the collection of `for` equations will not be taken into account for subsequent benchmarks.

Hence, the basis for the presented method is a flat DAE including both scalar variables and unexpanded array-variables which have to be matched to either single equations or `for` equations. Figure 4 depicts an overview of the procedure.

Classify equations: In order to reduce the problem size, all DAE-equations are classified into the three groups *for-equations* (containing only unexpanded array-variables), *non-array-equations* (containing only non array-variables), and *mixed-equations* (containing scalar array-variables). Variables are classified as either array-variables or non-array-variables.

Removal of trivial equations: The removal of trivial equations is comprised of both the removal of simple equations like `a = b`, `a = -b`, or `a = const` and a replacement of the respective alias variables to maintain the balance of equations and variables. This is a common optimization in model compilation in order to reduce the system size. Trivial equations have a prominent fraction among all model equations (from 44% to 73%) so this optimization is an eminent operation for an efficient model compilation. The scalar implementation for removing of trivial equations is applied on the *mixed* and *non-array* equations. Trivial equations do occur in *for-equations* as well and the removal and replacement of alias variables can be adopted. The scalar implementation has to find `N` trivial equations for `N` repeated equations whereas the implementation for the compact `for` equations only has to detect a single trivial assignment to assign `N` alias variables.

Partial slicing of ranges: The goal of the upcoming matching is to assign variables explicitly to all equations. It is settled that a scalar equation can only solve a scalar variable and an `N`-dimensional `for` equation can only solve an `N`-dimensional array-variable. Therefore, it has to be ensured that every `for` equation is connected to array-variables of the same dimension range. Besides that, it is possible, that a certain scalar variable is solved in one of the *mixed-equations* or that it has to be replaced by its alias variable. Hence, the unexpanded array-variables have to be sliced in ranges of their com-

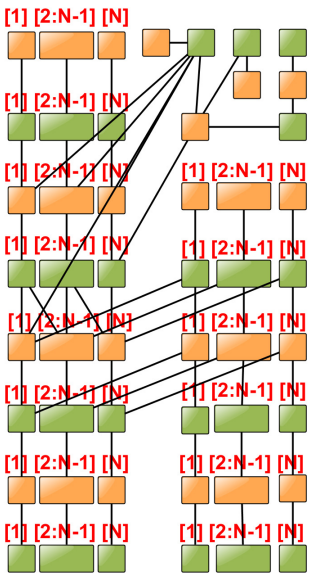


Figure 5. Bipartite graph with balanced dimensions.

plete dimension and single scalar array elements. First of all, the scalar array variables which have been replaced and the scalar array variables which occur in the *mixed-equations* have to be sliced from the unexpanded array. As a second step, a balance of dimension ranges between `for` equations and adjacent array-variables has to be established in the bipartite graph. This is performed by traversing all array-variables and compare the dimensions of the adjacent `for` equations. It has to be checked whether the dimensions have to be adjusted by slicing scalar equations or variables. Figure 5 depicts the partially sliced system of the wire model. As can be seen, every `N`-dimensional equation is connected to `N`-dimensional array variables (or scalar variables which have to be matched to a scalar equation instance).

Matching: If the bipartite graph is completely balanced with respect to the dimensions, matching can be performed as usual. Every equation node with only one adjacent variable node will be matched to this particular variable node. Thus, a matched variable can be taken as known and all edges can be removed from the graph. The order of matching assignments corresponds to the computation sequence. The current implementation offers a basic compilation of models containing repetitions. At the moment, no index-reduction algorithm is realized and no algebraic loops are allowed to occur. Furthermore, interruptions in the iteration space are not yet handled. However, these features are usually not required in electrical and hydraulical transmission elements. When compiling the electrical wire model, an ODE system is generated which contains a constant number of equations. Only the number of iterations inside the `for` loops is dependent on the number of discretized elements.

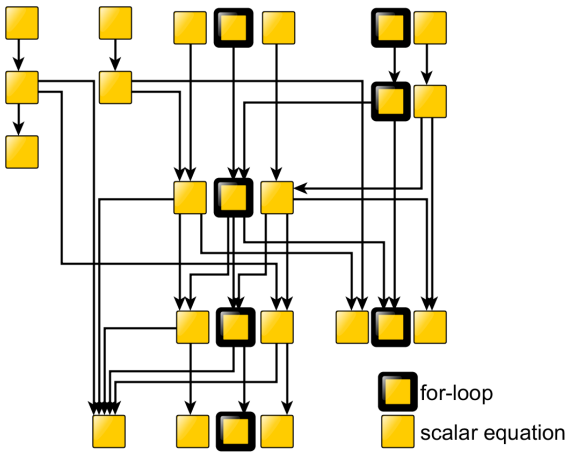


Figure 6. Task graph representation of causal ODE system with compact `for`-equation notation.

2.2 Results

Figure 6 depicts the task-graph of the causalized system for the wire model. A task graph represents the computation sequence of equations and algebraic loops which are displayed as nodes connected by edges refering to data dependencies (Walther et al., 2014). A node depicts a single equation to be solved in order to compute the successive equations. A bold task represents a vector task containing a `for` loop.

As can be seen, the compact notation is preserved throughout symbolic transformation. This results in faster compilation due to a reduced problem size. Figure 7 shows the number of equations which have to be processed in the compiler. By using the compact `for` notation, the number of equations can be kept at a constant size for different numbers of repetitions. Applying the removal of trivial `for` equations and subsequent scalarization of the DAE reduces the problem size significantly, as is shown with the green bars.

Figure 8 shows the time to translate the model from a flattened DAE-system to a solvable system. Obviously, the symbolic transformation for the default compilation of a flattened DAE-system does not scale linearly. The translation using the compact `for` notation results in a nearly constant compilation time. In order to reveal the advantages by using `for` notation for the removal of trivial equations, a third bar is depicted. The green bars show the compilation time if the removal of trivial equations has been performed on the `for` equations and the system is scalarized completely afterwards. This can be considered the minimal solution to exploit repeated equations.

The task graph representation is the basis for the generation of program code. Like in Modelica, `for` loops are a first level language concept in C and C++. By adopting the code generation to the new `for` loop tasks, the size of the generated code and the compiled executable are reduced, as depicted in Figure 9.

DOI 10.3384/ecp15118265

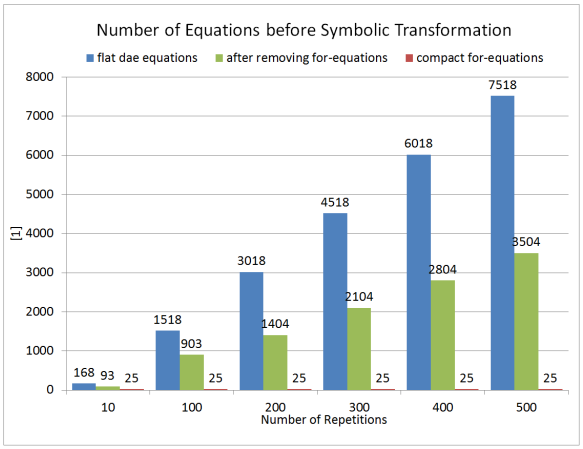


Figure 7. Comparison of number of equations to process the symbolic transformation for the default implementation, the vectorized compilation and completely scalarized system after removing of trivial `for` equations.

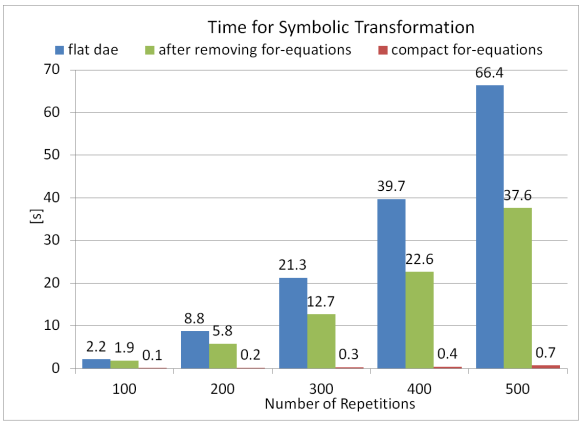


Figure 8. Comparison of symbolic transformation time between default implementation, complete vectorized compilation and completely scalarized system after removing of trivial `for` equations.

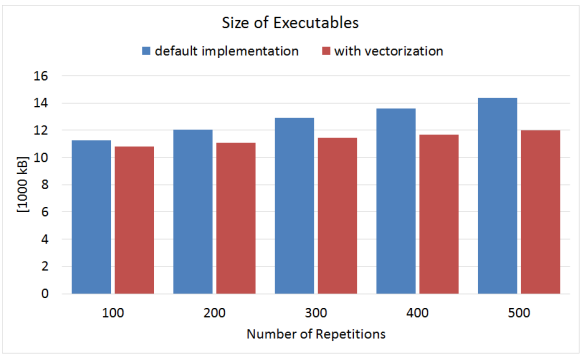


Figure 9. Comparison of executable size between default implementation and vectorized compilation.

The difference in size of the executables is little since the declaration of system equations is only one small part of the overall program code. Nevertheless, it is a an evident implication of the compact `for` loop notation.

The wire model does not contain algebraic loops or complex function calls. Hence, it can be simulated very fast which is why no simulation time comparison has been done. The benefit of vectorization for improving simulation times will be presented in the Section 4.

3 Causal For Loop Statements in Modelica

Loop statements in Modelica models can both occur in algorithm and equation sections. In algorithm sections like Modelica functions, they are trivially passed through the model compilation since they do not have to be causalized or manipulated. Therefore, causal `for` loop statements can be forwarded directly to the code generation and vectorization can be applied. This can lead to improved simulation performance so using `for` loops should be a common modeling best practice.

One exemplary model is `Fluid.Examples.BranchingDynamicPipes` from the Modelica Standard Library 3.2.1. This model makes heavy use of property computations defined in the Media library. Various Modelica functions are defined that compute the property values for the employed medium. One function used when simulating moist air is `Media.Air.MoistAir.saturationPressureLiquid`. The default implementation to calculate the saturation pressure is:

```
psat := exp(((
          a[1]*r1^n[1]
        + a[2]*r1^n[2]
        + a[3]*r1^n[3]
        + a[4]*r1^n[4]
        + a[5]*r1^n[5]
        + a[6]*r1^n[6])
        * Tcritical) / Tsat) * pcritical;
```

Since a major part of this expression consists of operations on contiguous memory elements, this computation can be vectorized. The function can be rewritten to compute the saturation by means of a `for` loop:

```
for i in 1:6 loop
  aux := aux + a[i]*r1^n[i];
end for;
psat := exp((aux * Tcritical) / Tsat)
        * pcritical;
```

This is just one example of how Modelica functions can utilize `for` statements. Due to the extensive use of `for` statements in this model, vectorization can have distinct effects on simulation performance, as will be explained in the following section.

4 Vectorization

In many cases, modern CPU architectures provide vector units that allow parallel execution of the same in-

struction on multiple data elements which is known as SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata). SIMD parallelization often requires less hardware for parallel computation than multiple full cores do. Hence, if used correctly, the exploitation of available SIMD instruction sets promises improved performance without requiring additional hardware. It also improves energy-efficiency, since more computation can be done per clock cycle and thus per Watt.

There are various ways of generating vector code. While low-level techniques, like vector intrinsics and assembly code (Intel), could in theory be employed in the code generation of a Modelica compiler backend, it requires much more caution and effort by the developers of the Modelica compilers. Recent developments on the C/C++ compiler infrastructures have brought powerful tools to developers that make it easier to exploit the computational power of vector units. Moreover, relying on the compiler to efficiently vectorize code also guarantees compatibility with future hardware, which otherwise would require extensions to the Modelica compiler in order to support new hardware platforms. Thus, the presented work focuses on automatic compiler vectorization in order to ensure correctness, portability, and ease of maintenance.

4.1 Compiler-based Vectorization

Vectorization is based on the SIMD principle, which requires the same instruction to be executed on different data points. This is usually the case, if the application makes use of loop control structures that iterate over a vector of data points. Modern compilers, e.g., the GNU compiler collection (GCC), the Intel compiler, and other compiler products and frameworks, incorporate sophisticated analysis and optimization logic to detect vectorizable loops and emit the respective instructions for the available hardware (Maleki et al., 2011). However, certain constraints have to be met in order to allow the compiler to correctly and efficiently vectorize a loop (Corden, 2012), including:

Absence of data dependencies: An iteration N of the vectorized loop may not consume the result of a previous iteration $N-1$ (read-after-write dependency). The same holds true for write-after-read dependencies, although modern compilers apply heuristics to detect vectorizable patterns such as reductions, e.g., in the example code in Section 3.

Aligned memory accesses: Modern CPU and memory architectures provide access to memory through caches, with the cache line size commonly at 64 B, which are loaded at once into the cache. Using non-unit-stride memory accesses might require loading multiple cache lines, which in turn could impact performance and thus outweigh the benefits of vectorization.

Linear iteration space and known step size: in order to correctly transform loops into vector statements, the step size and trip count have to be known at least at runtime. This excludes infinite loops, loops that are not bound by an explicit index variable, as well as loops with early exits.

Restricted function calls: Calling functions inside loops prevents the compiler from properly vectorizing the loop, unless these functions can be transformed into vectorized code, e.g., transcendental functions such as `pow()` or `exp()`, or functions that have been otherwise marked as vectorizable, e.g., using OpenMP SIMD statements (OpenMP, Sect. 2.8.2).

Avoiding branches: Branches commonly create conditionally diverging code paths which cannot be vectorized easily. With today's larger vector units, the compiler might be able to use mask registers to restrict operations to parts of a vector. However, branches should be avoided in general for best performance.

4.2 Modern Vector Architectures

Modern x86 CPUs are equipped with 128 up to 256-bit wide vector units and support for the Advanced Vector eXtensions (AVX and AVX2), which enable the parallel computation of up to four double or eight single precision floating point values. Other architectures, e.g., the ARM architecture with its NEON instruction set, also support vector instructions.[1] Without actually using these instructions, developers only make use of a fraction of the theoretical peak performance available. Upcoming generations of Intel CPUs will provide support for the AVX-512 instruction set, which once more doubles the width of the vector registers to 512 bit or eight parallel double precision floating point operations in parallel (Intel).

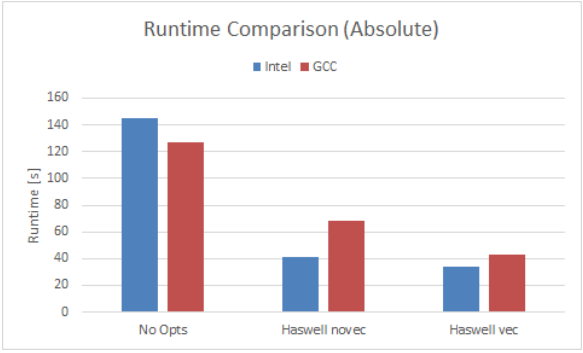
4.3 Results

As described in Section 3, several functions from the Modelica Standard Library that are used by the model `BranchingDynamicPipes` to contain vectorizable loops have been adjusted. These loops make heavy use of the `pow()` function, which has been vectorized by the compiler.

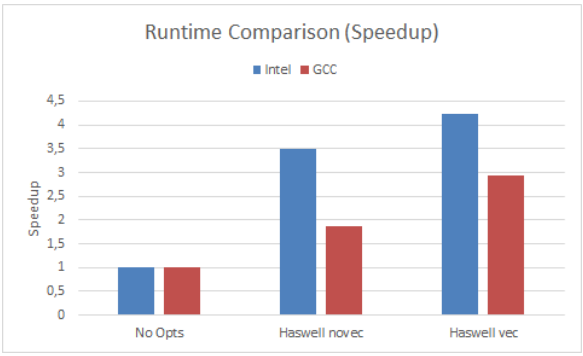
To compare the effects of vectorization on the runtime, we compiled the model into C++ code and translated it using different compiler settings (see Table 1). Since the modifications to the Modelica code were minor (combining unrolled loops into for statements with short trip

[1] `http://www.arm.com/products/processors/technologies/neon.php`, accessed 2015-05-19

DOI 10.3384/ecp15118265



(a) Absolute runtimes



(b) Relative Speedup

Figure 10. Resulting runtimes and speedup without optimizations, with standard compiler optimizations and with vectorization for different compilers.

counts), the behavior of the code did not change. The difference in performance between a loop and its unrolled counterpart are negligible if the loop body is computationaly heavy, as is the case with the `pow()` function. Hence, changing the compiler flags that control the vectorization is sufficient for a fair comparison.

It should be noted that none of the compilers used actually perform vectorization if optimizations are disabled (`-O0`). Since this is the default configuration in OpenModelica, measurements for runs without any optimization are included, too.

The following measurements were performed on a single socket system equipped with 8 GB of RAM and an Intel Core i7-4770 (Haswell microarchitecture) eight core CPU with HyperThreading disabled. The CPU supports the AVX2 instruction set mentioned in Section 4.2. The system was running an up-to-date Ubuntu GNU/Linux version 14.04 with kernel 3.13.0-45-generic.

The differences in runtime between runs using no optimization, optimization level two (`-O2`) without vectorization and optimization level two with vectorization are presented in Figure 10a. The corresponding relative speedups are displayed in Figure 10b. The difference between a run without optimizations and with default optimizations is significant, ranging from 1.9 for the GNU compiler to 3.4 for the Intel compiler. By enabling vectorization, the code compiled with the GNU compiler provides a reduction in runtime of 37.5 % on top of the default level two optimizations and a factor of 2.95 compared to the run without any optimizations. For the Intel compiler, the speedup gained by enabling level two optimizations is much more significant, around factor 3.5. However, the speedup from vectorization is only 20 % on top of that, totaling to a factor of 4.23.

It should be noted that only a fraction of the code has been vectorized, namely the functions from the Media library that form the most compute intensive kernels in this model (see Section 3). On top of that it should be noted that for all optimized runs the FMA feature had to be disabled. Fused-Multiply-Add (FMA) is an instruction that combines one multiplication and one addition, avoiding rounding of intermediate results and potentially doubling the arithmetic throughput. However, it also has been observed to impact the numerical behavior of the application, potentially increasing run-times of the solver component.

5 Conclusion and Outlook

The presented work aims to motivate the use of `for` loops as a modeling best practice in Modelica. The benefits from exploiting these repeated structures have been demonstrated. Passing `for` loop constructs through the compilation process accelerates the symbolic transformation, the program compilation as well as the model execution. The size of the generated code and executable can be reduced. Furthermore, it also enables the utilization of vector-units to perform SIMD operations effectively. The simulation time can be reduced clearly when applying automatic vectorization on `for` loops.

The demonstrated loop-preserving compilation features are still limited in their functionality. First of all, an instantiation providing compact `for` equations would replace the costly collection of similar equations which is currently performed in the OpenModelica Compiler backend. In order to extend the range of manageable models, the implementation has to be extended to support index-reduction and algebraic loops. Also the interruption of repeated structures, nested-loops, or non-linear iterations are potential research topics. Moreover, loop fusion is another worthwhile research target that can provide higher arithmetic density by reusing intermediate values and improving vectorization efficiency.

As has been mentioned earlier, certain features of modern CPU instruction sets can lead to small numerical differences in the numerical behavior of the application, leading to significant consequences for the total simulation performance. Further investigations will be conducted to allow the full exploitation of available hardware features while avoiding increased simulation times.

	GCC	Intel
Version	4.9.2	15.0.3 20150407
	Optimization Flags	
No Opts	-O0	-O0
Haswell novec	-O2 -fno-tree-loop-vectorize -march=haswell -mno-fma -ffast-math	-O2 -no-vec -no-simd
Haswell vec	-O2 -ftree-loop-vectorize -mno-fma -ffast-math -mveclibabi=svml -march=haswell	-O2 -xCORE-AVX2 -fno-fma

Table 1. Compiler versions and flags overview.

Acknowledgments

The presented work is part of the HPCOM project, that is funded by the Federal Ministry of Education and Research (BMBF) under the support code 01IH13002B. Parts of the work have been funded by the Intel Parallel Computing Center at the Center for Information Services and High Performance Computing at the TU Dresden.

References

Matthias Arzt, Volker Waurich, and Jörg Wensch. Towards Utilizing Repeating Structures for Constant Time Compilation of Large Modelica Models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '14, pages 35–38, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2953-8. doi:10.1145/2666202.2666207. URL http://doi.acm.org/10.1145/2666202.2666207.

Martyn Corden. Requirements for Vectorizable Loops, 2012. URL https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops. Accessed 2015-05-19.

Christoph Höger. Separate Compilation of Causalized Equations -Work in Progress. *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2011, Zurich, Switzerland, September 5, 2011*, pages 113–120, 2011.

Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. Intel, October 2014. URL https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf. Accessed 2015-05-19.

Saeed Maleki, Yaoqing Gao, Maria J. Garzaran, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, Oct 2011. doi:10.1109/PACT.2011.68. URL polaris.cs.uiuc.edu/~garzaran/doc/pact11.pdf.

OpenMP. *OpenMP Application Program Interface*. OpenMP Architecture Review Board, Jul 2013. URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

Constantinos C. Pantelides. The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988. doi:10.1137/0909014. URL http://dx.doi.org/10.1137/0909014.

Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing.*, pages 146–160, 1972. URL langevin.univ-tln.fr/cours/PAA/extra/Tarjan-1972.pdf.

Marcus Walther, Volker Waurich, Christian Schubert, Ines Gubsch, Andreas Hofmann, and Lars Mikelsons. Equation based parallelization of Modelica models. In *Proceedings of the 10th International Modelica Conference*, 2014.

Dirk Zimmer. Module-Preserving Compilation of Modelica Models. *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, (2):880–889, 2009.