Smart Processing of Function Calls to Achieve Efficient Simulation Code

Jan Hagemann Patrick Täuber Lennart Ochel Bernhard Bachmann

Department of Engineering and Mathematics, University of Applied Sciences Bielefeld, Germany,
{jan.hagemann,patrick_marcel.taeuber,lennart.ochel,bernhard.bachmann}@fh-bielefeld.de

Abstract

This paper introduces a new algorithm to increase the simulation performance of algebraic equation systems by encapsulating function calls. This avoids unnecessary evaluations of function calls and leads to positive structural effects, such as code motion. To enable the reader to reconstruct the algorithm, all four phases of the algorithm are described in detail and the complexity of them is analyzed. The overall complexity for practical models is $O(n)$, where n is the number of equations. It is shown that the algorithm significantly decreases the simulation time for a wide range of physical based models.
Keywords: function calls, backend, simulation

1 Introduction

Symbolic transformation and simplification methods (e.g. alias elimination, tearing methods, and index reduction) are essential within a Modelica compiler in order to achieve efficient simulation of the underlying differential-algebraic equation system. In this paper a proper handling of time-consuming function calls is discussed. It will be shown that the corresponding implementation of the module *WrapFunctionCalls* results in a tremendous decrease of simulation time for many models of specific libraries including the *MSL 3.2.1*. The expected performance increase has been already discussed in earlier publications (Jorissen et al., 2015).

The module *WrapFunctionCalls* traverses the equation system for occurring function calls. Those are stored in temporary variables, which are inserted at the according places in the equation system. This elimination is similar to the *Common Subexpression Elimination* (Jakubowski, 2002), hence the auxiliary variables for the function calls are called CSE-variables. However, the difference is that in the case of *WrapFunctionCalls* also single occurring function calls are stored.

There are some requirements on an efficient and reliable algorithm to encapsulate function calls, which are discussed in the next section before the algorithm is presented in Section 3. Such or similar symbolic transformations of function calls could not be found in literature or are not accessible in other simulation environments.

At the end of the paper a verification section proves the effects on models with a practical orientation and therefore the significance of the algorithm.

2 Requirements on the Algorithm

To achieve efficient simulation code by processing function calls first and foremost all function calls must be found regardless whether they occur in simple equations or if the function itself is an argument to another function. Consequently, nested function calls (e.g. $exp(cos(time))$) must be analyzed in detail to guarantee that function calls in arguments of other functions are replaced as well.

At this point it should be mentioned that only equations are traversed for functions. Algorithms are not handled by the module yet. Special Modelica constructs like impure functions are not traversed for nested calls as well and functions which are called conditionally (i.e. functions in bodies of when- and if-equations) are not encapsulated if they do not appear in the rest of the equation system.

All the found functions must be stored in CSE-variables. Considering the Modelica modelling language, it must be noted that for calls in equations of the form

$$variable = call \quad \text{(e.g. } x = cos(time)\text{)}$$

or

$$tuple = call \quad \text{(e.g. } (a,b,\_) = f(time,x)\text{)}$$

no new CSE-variables should be introduced, because in those cases the variables on the left-hand side already encapsulate the corresponding functions and can therefore be used as CSE-variables.
Additionally, identical function calls, which do not necessarily have to look similar at the first sight, have to be recognized. The following example shows three equations with mathematically identical function calls in the second and third equation, due to the first equation:

$$\begin{aligned} x &= cos(time) \\ a &= exp(x) \\ y &= exp(cos(time)) \end{aligned}$$

The main objective of the algorithm is to reduce the simulation time by encapsulating function calls. Thus, duplicated calls only need to be evaluated once instead of every time they occur in the equation system. Especially for models with more complex functions, this will lead to vast

time savings during simulation.
Nevertheless, this is not the only way to reduce expensive function evaluations. As already mentioned above, one difference to the well known *Common Subexpression Elimination* methods is that function calls that only occur once in the whole model are encapsulated as well, because this can lead to a desirable code motion effect. Code motion has its origin in the optimization of standard compilers (Aho et al., 2008; Muchnick, 1997; Vogt, 2004). The code expressions, which appear in loops (e.g. for, while, ...) and are independent of the iteration variables and their dependencies can be moved out of the loop and be evaluated beforehand. In Modelica code motion can be seen as the extraction of subexpressions, i.e. function calls, out of algebraic loops. The following call illustrates the advantage of code motion in an acausal environment by encapsulating all function calls:

$$f_1(f_2(x), f_3(time))$$

Assuming this call to be part of an algebraic loop, where x may be the iteration variable, by encapsulating the calls in the following way, the calculation of function f_3 can be moved out of the loop because it is not dependent on variables, that are changing its value during the iteration process:

$$cse_1 = f_1(cse_2, cse_3)$$
$$cse_2 = f_2(x)$$
$$cse_3 = f_3(time)$$

Without the encapsulation f_3 would be evaluated in each iteration step, which could be expensive.

The verification of the expected effects mentioned above is done in Section 4.

3 Algorithm

The algorithm *WrapFunctionCalls* is divided into four parts:

I. Analysis
II. Dependencies
III. Substitution
IV. Creation of CSE-Equations

For a better understanding of the functionality the algorithm is introduced section by section with an example and the corresponding pseudo code. After each part the complexity is assessed.
Listing 1 shows the abstract example model that is used to illustrate the requirements on a reliable algorithm. Thus, not only the sine and cosine function calls must be stored in CSE-variables but also the more complex function foo, which has more than one return value, that must be handled properly. The function foo occurs in two equations. In the first one, only the second return value is accessed. In the second equation, however, only the first output is used. Additionally, both calls of foo are identical because of the third equation. Lastly, it must be considered that the top-level functions in the first and third equation do not need to be stored in CSE-variables, because the left-hand side is already a simple variable.

```
1  model wfc
2    function foo
3      input Real x1;
4      input Real x2;
5      output Real y1;
6      output Real y2;
7      output Real y3;
8      [...]
9    end foo;
10   Real a, b, x;
11 equation
12   (,b,) = foo(x, sin(cos(time)));
13   a = sin(foo(x, x)) + 5.0;
14   x = sin(cos(time));
15 end wfc;
```

Listing 1. Abstract example model to introduce *WrapFunctionCalls*

The algorithm works with three data structures, which need to be created at the beginning. First, a hash table is needed to store and access the found function calls efficiently. Due to expiriences with the *MSL* the size of the hashtable is chosen accordingly to avoid collisions. The keys of the hash table are the function calls and the values are integers representing indices of an expandable array, which is the second data structure. The expandable array contains entries consisting of a CSE-variable, a function call and an integer list, which represents the dependencies between function calls. As third data structure, another array is needed to store the manipulated equations and the new CSE-equations (Listing 2).

```
1  // Array of equations: eqArray
2  // Equation: eq
3  // List of variables of eqSystem: varList
4  // Expression: exp
5  // Subexpression: subExp
6  // CSE variable: cse_var
7  // Hash table: ht
8  // Expandable array: array+
9
10 create ht :=
11   <key=call(exp), value=index(int)>;
12
13 create array+ :=
14   [cse_var(exp), call(exp), dependencies(list<
       int>)];
15
16 create eqArray := [];
```

Listing 2. Pre-phase of *WrapFunctionCalls* algorithm: Create data structures

3.1 Analysis

In the first part of the algorithm all equations of the equation system are traversed top-down to collect all function calls. A detected function call is stored within the

DOI 10.3384/ecp17132581

hash table, together with an index referring to the correspondent entry of the expandable array, which is basically an incremented integer for each call starting by 0. The corresponding entry of the expandable array contains the CSE-variable, the function call itself and an initially empty dependency list. In the first part of the algorithm all function calls are stored, even those which might turn out to be identical to other ones later on. To avoid the introduction of unnecessary CSE-variables an additional conditional branch is needed to detect equations of the form $var = call$ and $tuple = call$. In that case the already existing variables are stored as CSE-variables in the expandable array. If in the expandable array a CSE-variable has been entered for a call that turns out to be equal to an existing variable or tuple, the CSE-variable entry is overwritten with the already existing variable (cf. Listing 3).

```
17  index = 0;
18  // I. Analysis of eqSystem
19  for each eq in eqSystem
20    if eq as (cref=call) or (tuple=call) then
21      if call has not an entry in ht then
22        create ht entry := <call, index>;
23        create array+ entry := [cref/tuple, call
              , {}];
24        index = index + 1;
25      else
26        update/merge array+ entry = [Cref/Tuple,
              Call, {}];
27      end if;
28    end if;
29
30    for each call in eq [TopDown]
31      if call has not an entry in ht then
32        create ht entry := <call, index>;
33        create a cse_var for call;
34        create array+ entry := [cse_var, call,
              {}];
35        index = index + 1;
36      end if;
37    end for;
38  end for;
39
40  if ht is empty then
41    // algorithm terminates
42    return;
43  end if;
```

Listing 3. Phase I of *WrapFunctionCalls* algorithm: Analysis

Since the operations depend on the number of equations and each equation is addressed exactly one time, the complexity of the first part of the algorithm is $O(n)$, where n is the number of equations in the equation system.

If the analysis is performed on the example model from Listing 1 the hash table from Table 1 is generated. Because of the top-down traversal, the first call detected is $foo(x, \sin(\cos(time)))$. After that $\sin(\cos(time))$ and $\cos(time)$ are found. The same applies to the second equation. In the last equation a call is found, which already exists in the hash table, so no new hash table entry is created.

The detected function calls are also stored in the expandable array at the corresponding position determined by the index from the hash table. For the first call no additional CSE-variable is necessary because the call is equal to a tuple, so the tuple is stored as CSE-variable. For all other calls CSE-variables are introduced. Since the call in the last equation is equal to x and the call already exists in the hash table, the CSE-variable for $\sin(\cos(time))$ is overwritten with x again. This leads to the expandable array from Table 2, where the integer dependency lists are still empty.

Table 1. Hash table after the first part of the algorithm (Analysis)

Hash Table (ht)	
Function Call	Integer
$foo(x, \sin(\cos(time)))$	1
$\sin(\cos(time))$	2
$\cos(time)$	3
$\sin(foo(x,x))$	4
$foo(x,x)$	5

Table 2. Expandable array after the first part of the algorithm (Analysis)

Array+		
CSE-Variable	Function Call	Integer List
$(\_, b, \_)$	$foo(x, \sin(\cos(time)))$	{ }
x	$\sin(\cos(time))$	{ }
cse_2	$\cos(time)$	{ }
cse_3	$\sin(foo(x,x))$	{ }
$(cse_4, \_, \_)$	$foo(x,x)$	{ }

3.2 Dependencies

At the beginning of the second phase all the occurring function calls are already located in the expandable array and in the hash table. In the second part the analysis continues by considering each array entry to find the dependencies between the different function calls. To do so, the function calls must be analyzed successively whether there are other function calls in their arguments. If another function call is detected within an argument, the index (value) of that call has to be determined from the hash table. Now, at the position of that index in the expandable array, the dependency list is appended by the index of the outer function call (Listing 4). So, at the end the dependency list for each call contains the indices of the calls in which the current call occurs. An empty dependency list after the second phase signifies a function call independent of all other function calls.

```
44  // II. Dependencies
45  for each entry of array+
46    traverse arguments of call
47    add index of call to each argument entry (in
          list<int>);
48  end for;
```

Listing 4. Phase II of *WrapFunctionCalls* algorithm: Dependencies

If no nested function calls occur in the equation system, the complexity of the second phase is $O(k)$, where k is the number of supposedly different function calls, i.e. the number of array elements because each array entry is addressed only once.

If there are nested function calls, the complexity is smaller than or equal to $O(k+\sum_{i=1}^{a} i) = O(k+\frac{a^2+a}{2}) = O(k+a^2)$, where a is the number of function calls in the arguments of other function calls in the model, because each call has to be addressed again for each occurrence in the arguments of another call. The worst case of $k+\frac{a^2+a}{2}$ operations pertains if there is exactly one nested function call in the equation system because this would lead, for instance, to the following entries in the array, where a is $k-1$:

$$
\begin{aligned}
&f_1(f_2(...(f_{k-1}(f_k(x)))...))\\
&f_2(...(f_{k-1}(f_k(x)))...)\\
&\vdots\\
&f_{k-1}(f_k(x))\\
&f_k(x)
\end{aligned}
$$

Since $\sum_{i=1}^{a_1} i + \sum_{i=1}^{a_2} i < \sum_{i=1}^{a_1+a_2} i$ less operations have to be performed if all the inner calls are not in only one call but distributed in different calls. In practical and realistic models the worst case would not occur for a big k, because a high level of nesting in functions is not used. An analysis of the models of the *Modelica Standard Library 3.2.1* shows that the average maximum dependency list length is about 0.5, where the fluid and media models have the largest nesting. Thus, with respect to the number of function calls the calls in arguments of other calls are negligible. This assumptions lead to a nearly linear complexity of $O(k)$.

Table 3. Expandable array after the second part of the algorithm (Dependencies)

Array+		
CSE-Variable	Function Call	Integer List
$(\_, b, \_)$	$foo(x, \sin(\cos(time)))$	$\{\}$
x	$\sin(\cos(time))$	$\{1\}$
cse_2	$\cos(time)$	$\{1, 2\}$
cse_3	$\sin(foo(x,x))$	$\{\}$
$(cse_4, \_, \_)$	$foo(x,x)$	$\{4\}$

Applying the second phase on the example leads to the dependency list from Table 3. Since $\sin(\cos(time))$ occurs in the first entry index 1 is entered in its dependency list. The cosine occurs in the first and second entry, so its dependency list is $\{1,2\}$, while $foo(x,x)$ appears in entry four and therefore has the dependency list entry 4. For the other function calls the dependency lists remain empty since they are the top-level functions and do not appear in any other function arguments.

3.3 Substitution

In the third part of the algorithm the equations in the equation system are traversed again but this time bottom-up. The encountered function calls are replaced with the CSE-variables. If the corresponding dependency list has entries, the function calls in the expandable array at which the indices point to are replaced as well and the dependency list is deleted afterwards. Additionally, a new entry in the hash table is created, which contains the new function call and its original index in the array, to guarantee that occurring calls of that kind can be found as well. If this entry is already existent in the hash table, then the CSE-variables have to be merged in the expandable array. After the processing of each equation the substituted equation is added to the equation array if it is not redundant. If the substituted equation is redundant, such as $x = x$, it is discarded (cf. Listing 5).

```
49 // III. Substitution
50 for each eq in eqSystem
51   for each call in eq [BottomUp]
52     get cse_var from array+ and substitute
         call;
53     substitute the arguments with the cse_var
         in the calls referenced by the
         dependency list;
54     delete the dependency list of the current
         cse_var;
55     if not already in ht then
56       add the new call to ht with the original
           index;
57     else
58       merge cse variables in the expandable
           array
59     end if
60     if the substituted eq is not redundant
         then
61       add eq to eqArray;
62     end if;
63   end for;
64 end for;
```

Listing 5. Phase III of *WrapFunctionCalls* algorithm: Substitution

Since the number of operations in the equation system depends on the number of equations, and the number of additional operations in the expandable array depends on the total number of all dependency list entries, which is negligible relative to the number of equations in models with a practical orientation, the complexity for the third phase is nearly linear regarding n ($O(n)$).

If the third phase is performed on the example model, the equations are traversed again. This time $\cos(time)$ is the first call that is detected because now the algorithm works bottom-up. So $\cos(time)$ is substituted by its CSE-variable cse_2. Additionally, the calls in the array the dependency list of $cos(time)$ points to are substituted as well,

DOI 10.3384/ecp17132581

so the first call is $foo(x, \sin(cse_2))$ and the second call is $\sin(cse_2)$ now. The dependency list is deleted because this substitution only needs to be performed once. Finally, new hash table entries are created for $foo(x, \sin(cse_2))$ and $\sin(cse_2)$ with the values 1 and 2, respectively.
The next call that is found is $\sin(cse_2)$, which is replaced by x in the equation and in the first array entry and its dependency list is deleted. The first array entry now contains $foo(x,x)$. Since this call already exists in the hash table, this call is now equal to another call and the introduced CSE-variables have to be merged. Therefore, the CSE-variables in the fifth array entry are changed to $(cse_4, b, \_)$. The function call in the first array entry is also overwritten with $(cse_4, b, \_)$.
After that, the call $foo(x,x)$ is found in the substituted equation. It is replaced by $(cse_4, b, \_)$ in the equation and by cse_4 in the forth array entry because there the first output is needed. The call $sin(cse_4)$ is added to the hash table with value 4 and the dependency list is deleted. At the end of the processing of the first equation the substituted equation is $(\_, b, \_) = (cse_4, b, \_)$. Since this equation is redundant it is not added to the equation array.
While processing the next equations, no more manipulations on the hash table and on the expandable array have to be performed. At the end of the substitution of the second equation it reads $a = cse_3 + 5.0$. It is added to the equation array. The third equation reads $x = x$ and is not added to the equation list because it is redundant.
The hash table and the expandable array after the third phase are illustrated in Table 4 and Table 5, respectively.

Table 4. Hash table after the third part of the algorithm (Substitution)

Hash Table (ht)	
Function Call	Integer
$foo(x, \sin(\cos(time)))$	1
$\sin(\cos(time))$	2
$\cos(time)$	3
$\sin(foo(x,x))$	4
$foo(x,x)$	5
$foo(x, \sin(cse_2))$	1
$\sin(cse_2)$	2
$\sin(cse_4)$	4

Table 5. Expandable array after the third part of the algorithm (Substitution)

Array+		
CSE-Variable	Function Call	Integer List
$(\_, b, \_)$	$(cse_4, b, \_)$	{ }
x	$\sin(cse_2)$	{ }
cse_2	$\cos(time)$	{ }
cse_3	$\sin(cse_4)$	{ }
$(cse_4, b, \_)$	$foo(x,x)$	{ }

3.4 Create CSE-Equations

In the fourth and last part of the algorithm each entry of the expandable array is considered and an equation with the CSE-variable and the corresponding expression is generated. If that equation is not redundant it is added to the equation array and the CSE-variable is added to the variable list (Listing 6).

```
65  // IV. Create cse equations
66  for each entry of array+
67    generate eq (cse_var=call);
68    if eq is not redundant then
69      add eq to eqArray;
70      add cse_var to varList;
71    end if;
72  end for;
73
74  add eqArray and varList to eqSystem;
```

Listing 6. Phase IV of *WrapFunctionCalls* algorithm: Create CSE-equations

Since each array entry is addressed one time, the complexity of this phase is $O(k)$.

In the example, all equations derived from the expandable array, except the one from the first entry, are added to the equation array. So the generated CSE-equations in the example are the following:

$$\begin{aligned} x &= sin(cse2) \\ cse2 &= cos(time) \\ cse3 &= sin(cse4) \\ (cse4, b, \ ) &= foo(x,x) \end{aligned}$$

The processed example model at the end of the *WrapFunctionCalls* algorithm is shown in Listing 7.

```
1  model wfc_result
2    function foo
3      input Real x1,
4      input Real x2;
5      output Real y1;
6      output Real y2;
7      output Real y3;
8      [...]
9    end foo;
10   Real a, b, x, cse2, cse3, cse4;
11 equation
12   a = cse3 + 5.0;
13   cse3 = sin(cse4);
14   (cse4, b, ) = foo(x, x);
15   x = sin(cse2);
16   cse2 = cos(time);
17 end wfc_result;
```

Listing 7. Example model after processing with *WrapFunctionCalls*

To estimate the costs of the complete algorithm the complexities of the single phases have to be considered together, so it adds up to $O(n+k+n+k)$. Since in practical models the number of different function calls is negligible relative to the number of equations, the complexity of the *WrapFunctionCalls* algorithm can be estimated with $O(n)$. The actual complexity is assessed in the next section.

4 Verification

4.1 Complexity

The complexity of the optimization module *WrapFunctionCalls* depends on the number of given function calls and equations as described above. The estimation of a complexity of $O(n)$ is confirmed by a test of the *SteamPipe* models of the *ScalableTestSuite* library (Casella, 2015a). The ScalableTestSuite library is developed by Francesco Casella and Kaan Sezginer, Politecnico di Milano (Casella and Sezginer, 2016). This includes different models among others of the electrical engineering, mechanical science and thermodynamics. The models are characterized by the fact that they are scalable by parameters, so as to test the ability of the Modelica tools in terms of an efficient compilation and simulation time as the size increases. To do so the execution time of the algorithm for these models was measured during compilation for different parameters N which is a scalable parameter depending linearly on the number of equations n.

The results of the tests are shown in Table 6 and Figure 1. These tests confirm that the execution time is linear proportionate to the parameter N and thus also linear to the number of equations n. Due to experiences with a variety of different other libraries it is assumed that this $O(n)$ bahaviour is not only valid for a certain structure of equations.

Table 6. Execution time of *WrapFunctionCalls* for SteamPipe models

Model	N	n	Calls	Time [s]
SteamPipe_N_10	10	262	40	0.0074
SteamPipe_N_20	20	522	80	0.0221
SteamPipe_N_40	40	1042	160	0.0298
SteamPipe_N_80	80	2082	320	0.0607
SteamPipe_N_160	160	4162	640	0.1233
SteamPipe_N_320	320	8322	1280	0.253
SteamPipe_N_640	640	16642	2560	0.552

4.2 Code Motion

Below it is shown that the expected effect of code motion actually happens in practice, i.e. function calls are moved out of algebraic loops, so that these are calculated beforehand and not in each iteration step. To do so the simulation time of different libraries are compared with the optimization module *WrapFunctionCalls* deactivated and activated, respectively. The results of the test are shown in the next section. The analysis shows that among others in the model *WaterIF97* of the *MSL 3.2.1* code motion occurs in practical relevant examples. This model has a nonlinear algebraic loop (Table 7), whose residual equation contains a function call that can be stored in a CSE-variable (*cse*4) and thus can be moved out of the loop (Table 8).

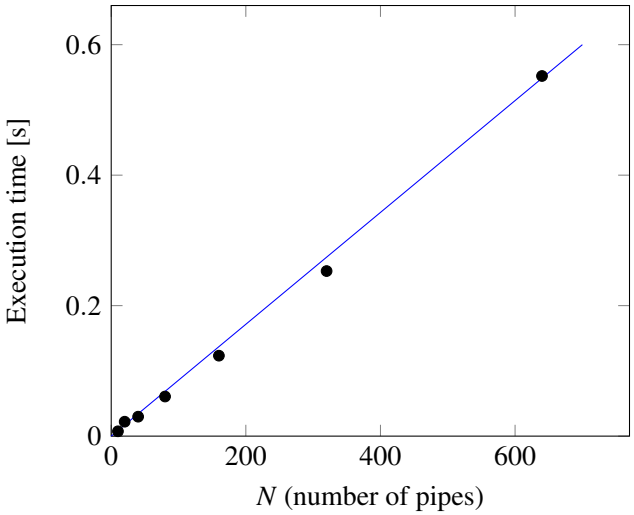

Figure 1. Graphic representation of the execution time of *WrapFunctionCalls* for SteamPipe models

Table 7. Code Motion example without *WrapFunctionCalls*

```
Nonlinear loop.
Iteration variable:  DER.medium.p
  DER.medium.h = DER.medium.u -
  (medium.p * DER.medium.d -
  der(medium.p) * medium.d)/(medium.d ^ 2.0)
Residual equation:
Modelica.Media.Water.IF97_Utilities.rho_ph_der
(medium.p, medium.h, Modelica.Media.Water.
IF97_Utilities.waterBaseProp_ph(medium.p,
medium.h, medium.phase, 0), DER.medium.p,
DER.medium.h) - DER.medium.d = 0.0
```

Table 8. Code Motion example with *WrapFunctionCalls*

```
cse4 := Modelica.Media.Water.IF97_Utilities.
  waterBaseProp_ph(medium.p, medium.h,
  medium.phase, 0);
```

```
Nonlinear loop.
Iteration variable:  DER.medium.p
  DER.medium.h = DER.medium.u -
  (medium.p * DER.medium.d -
  der(medium.p) * medium.d)/(medium.d ^ 2.0)
Residual equation:
Modelica.Media.Water.IF97_Utilities.rho_ph_der
(medium.p, medium.h, cse4, DER.medium.p,
DER.medium.h) - DER.medium.d = 0.0
```

4.3 Improvement in Simulation Time

This section shows the improvement in simulation time of many models in specific libraries. For this purpose, Table 9 presents some extracts of single libraries with improvements of over 40 percent. The reasons for the significant upgrade are especially the described effects of code motion and the encapsulation of expensive function calls.

DOI 10.3384/ecp17132581

The analysis shows that the *ThermoPower* library achieves the best results with an improvement of over 90 percent, followed by the *SteamPipe* models of the *ScalableTestSuite* with more than 80%. The *MSL 3.2.1* also demonstrates enhancements far above 40 percent. For example, the already mentioned model *WaterIF97* gains a speed-up of 53%.
These absolutely positive results are confirmed by several OpenModelica power users, which also demonstrate a magnificent improvement in runtime performance of their applications by encapsulating function calls. (Franke et al., 2015; Franke, 2016; Casella, 2014, 2015b).

Table 9. Improvement in simulation time shown on an extract of expressive libraries

Model (**Library**)	− WFC [s]	+ WFC [s]	Imp. [%]
MSL 3.2.1			
DrumBoiler	4.52	1.73	62
MomentumBalanceFittings	4.01	1.63	60
HeatExchangerSimulation	36.31	12.89	65
InverseParameterization	78.81	41.03	48
NonCircularPipes	8.71	3.91	55
R134a1	16.77	5.19	69
DryAir2	21.85	5.17	76
TestTwoPhaseStates	2.38	0.93	61
WaterIF97	1.90	0.90	53
PlanarMechanics			
KinematicLoop_ DynamicStateSelection	19.26	6.12	68
PowerSystems			
PowerWorld	7.59	1.73	77
ThermoPower			
CISESim180504	284.6	20.24	93
TestMixerSlowFastSteam	8.80	0.69	92
TestWaterFlow1DFV_F	45.73	15.82	65
ScalableTestSuite			
SteamPipe_N_10	23.84	4.35	81
SteamPipe_N_20	45.56	8.60	81
SteamPipe_N_40	98.24	17.61	82
SteamPipe_N_80	197.93	35.39	82
SteamPipe_N_160	412.87	73.8	82

5 Conclusions

This paper shows an optimization algorithm which encapsulates function calls of the given equation system in temporary variables. Above all, the idea is that expensive function calls should be evaluated only once. Furthermore, the effect of code motion of standard compilers motivates to find this effect in Modelica models. The extraction of function calls out of algebraic loops promises a huge improvement in simulation time since the iteration does not have to be performed over a function call in each step. Thus, also single occurring function calls are stored in contrast to the original *Common Subexpression Elimination*. Other peculiarities, e.g. nested function calls, functions with multiple outputs or simple equations with a special form such as $variable = call$, are considered as well.
Additionally, the algorithm works very efficient, which is confirmed by a complexity assessment of $O(n)$. A test with the help of the *ScalableTestSuite* approves the estimation. The desired effect of code motion also can be found in practical models and is depicted by an example of the *MSL 3.2.1*.
Moreover, single extracts of different libraries are listed, which show vast improvements in simulation time due to code motion and the encapsulation of function calls.

Acknowledgments

This work is supported by the Ministry of Innovation, Science and Research of the German State of North Rhine-Westphalia (MIWF NRW) as part of the research cooperation “MoRitS - Model-based Realization of intelligent Systems in Nano- and Biotechnologies" (grant no. 321 - 8.03.04.03 - 2012/02).

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools; 2. Edition*. Pearson, 2008. ISBN 978-0321486813.

Francesco Casella. Thermopower library simulation. OpenModelica Workshop, Linköping, Sweden, Februar 3rd, 2014, 2014. URL `http://www.modprod.liu.se/openmodelica2014-talks/1.544931/OpenModelica2014-talk03-Francesco-Casella-ThemoPowerLibrarySimulation.pdf`.

Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 459–468. Linköping University Electronic Press, Linköpings universitet, 2015a.

Francesco Casella. Modelling of energy systems with OpenModelica. OpenModelica Workshop, Linköping, Sweden, Februar 2nd, 2015, 2015b. URL `http://www.modprod.liu.se/openmodelica-2015/1.620217/OpenModelica2015-talk03-Francesco-Casella.pdf`.

Francesco Casella and Kaan Sezginer. ScalableTestSuite homepage. `https://github.com/casella/ScalableTestSuite`, December 2016.

Rüdiger Franke. Embedded optimizing control using the OpenModelica C++ runtime. OpenModelica Workshop, Linköping, Sweden, Februar 1st, 2016, 2016. URL `http://www.modprod.liu.se/filarkiv/1.672872/OpenModelica2016-talk08-RudigerFranke-EmbeddedOptimizingControl.pdf`.

Rüdiger Franke, Marcus Walther, Niklas Worschech, Willi Braun, and Bernhard Bachmann. Model-based control with FMI and a C++ runtime for Modelica. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, pages 339–347, 2015.

Jacek Jakubowski. Architekturunabhängige Quellcodeoptimierung durch Mustererkennung. Dissertation, Universität Dortmund, 2002.

Filip Jorissen, Michael Wetter, and Lieve Helsen. Simulation speed analysis and improvements of modelica models for building energy simulation. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, pages 59–69, 2015.

Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997. ISBN 978-1-55860-320-2.

Michael Vogt. Plattformunabhängige Eliminierung gemeinsamer Teilausdrücke auf Quellcode-Ebene. Dissertation, Universität Dortmund, 2004.

DOI 10.3384/ecp17132581