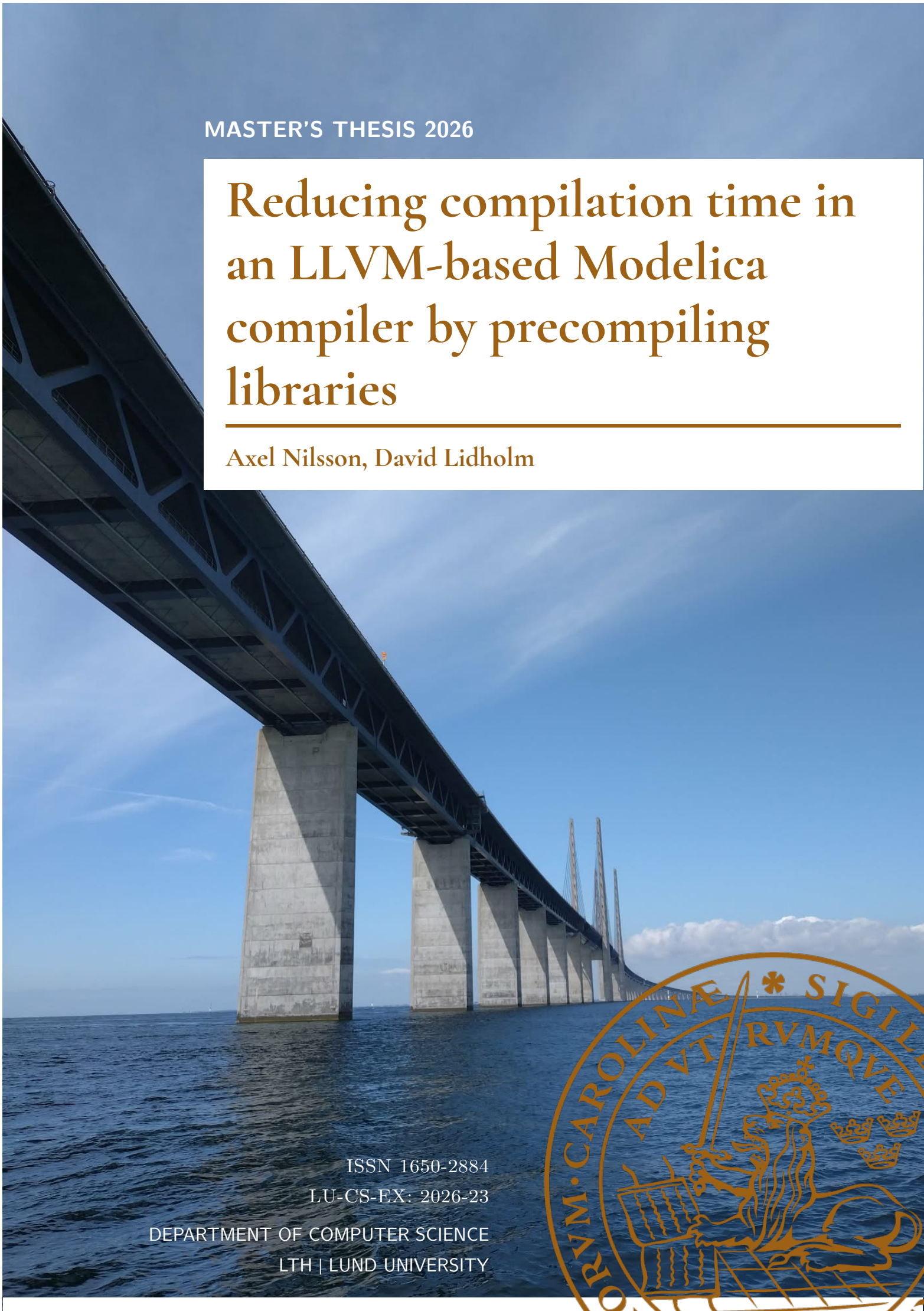


MASTER'S THESIS 2026

Reducing compilation time in an LLVM-based Modelica compiler by precompiling libraries

Axel Nilsson, David Lidholm

ISSN 1650-2884
LU-CS-EX: 2026-23

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-23

Reducing compilation time in an LLVM-based Modelica compiler by precompiling libraries

Minska kompileringstiden i en LLVM-baserad Modelicakompilator genom förkompilering av bibliotek

Axel Nilsson, David Lidholm

Reducing compilation time in an LLVM-based Modelica compiler by precompiling libraries

Axel Nilsson
axel.nilsson98@outlook.com

David Lidholm
lidholm02@gmail.com

June 16, 2026

Master's thesis work carried out at Modelon AB.

Supervisors: Filip Stenström, filip.stenstrom@modelon.com
Markus Olsson, markus.olsson@modelon.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Christoph Reichenbach, christoph.reichenbach@cs.lth.se

Abstract

This thesis investigates the potential of precompiling Modelica functions for a reduction in compile time when compiling Modelica models. Precompilation of functions was done in two ways:

1. Replacing the Modelica functions with external calls to equivalent and compiled C functions.
2. Replacing the Modelica functions in the backend of the compiler.

The second approach is in itself split into two different approaches where one filters the function from existing models and the other is doing an analysis of Modelica libraries to find functions.

The first approach failed to yield any conclusive results, but the second approach generated valid libraries that could be used with our test models. The optimization was tested on 6 different test models to evaluate performance compared to the baseline.

In general, the compile time reduction through utilization of precompiled functions has a significant dependence on the model's properties. Some models had a small increase in compile time, due to overhead introduced by our optimization, while others had up to 29.1% reduction in compile time at the backend. Additionally, the runtime was decreased due to optimizations of the precompiled functions for some models, with a reduction of up to 10.7% in the best case and about the same performance for the worst case.

Keywords: Compiler optimization, Precompilation, Modelica, LLVM

Acknowledgements

We thank our supervisors at Modelon; Filip and Markus for aiding us when the code base did not work like we wanted it to and helped to steer us back on track when we went off in the wrong direction. We also want to thank our supervisor at LTH, Jonas Skeppstedt, for his work at educating us in the world of compiler optimizations and aiding us in the process of writing this thesis.

Additionally, we want to thank our close friends and families for not giving up on us during our five years at LTH, but always suppoting us when we needed it.

Lastly, thanks to Olof for giving us the opportunity to capture the cover photo.

Contents

Chapter 1
Introduction

When designing complex systems, such as the cooling system of a car, creating a simulation of the system's behavior is crucial for reducing the development cost and time. To model such kinds of systems, an object-oriented modeling language called **Modelica** can be used. It simplifies the process of defining the equation systems controlling the simulation. To run the simulation of a system written in Modelica, the Modelica model must be translated into a form that can be executed[1]. There exist several tools that can translate, or **compile**, Modelica code into an executable format.

Modelon Impact, provided by **Modelon AB**, is one of the tools which can compile and execute Modelica models. It compiles the Modelica code into an executable, which can be used for simulation of the model. When using Modelon Impact, the compilation of the Modelica code can be very time consuming, which will slow down iteration time for developers when they are iterating in the design of their product since most iterations require a recompilation of the entire Modelica model. Therefore, reducing the compilation time is a priority for Modelon since customers must wait for the compilation and simulation to finish before they can continue working. Iteration time is dependent on both compilation time and simulation time, making this a trade-off in how much time can be spent optimizing code during the compilation to get the fastest iteration time.

Compilation of a Modelica model results in an equation system. This equation system is dependent on how different components in the model are connected. For example, if one uses a resistor from a library, the way the resistor connects to the rest of the circuit affects the resulting equation system. Thus, the variability in the equation systems makes it practically impossible to precompile entire Modelica libraries. However, there are parts in Modelica libraries that are independent of the equation structure. **Thus, a possible compiler optimization is explored in this thesis that is centered around the concept of separate compilation, where functions from Modelica libraries are precompiled into a format which can be linked during the construction of the executable. The thesis will investigate the effects on compile time and runtime performance when using this optimization.**

The reduction in compilation time could improve the productivity of product developers

that rely on Modelica for modeling and thus speed up the development of new products. Additionally, precompiling code enables compiler optimizations since the precompilation of libraries is done before the library is delivered to the user and not when the user is waiting to run a simulation. These optimizations are not done in the Modelon compiler due to compile time constraints. Therefore, using optimizations on the precompiled code should increase the performance of the simulation process.

A possible downside with separate compilation is that, since the compilation scope is smaller, the compiler might have less opportunity to perform certain optimizations, for example inlining of functions. This could perhaps affect the simulation time of our test negatively.

1.1 Research Questions

The research questions that are the basis of this thesis are the following:

- How can Modelica functions be extracted and precompiled?
- How does precompilation of functions affect runtime and compile time performance?
- When is it not possible to precompile a Modelica function?

1.2 Contributions

The results of this thesis will provide additional information that can be used in the process of improving the compile time performance of the Modelon compiler and the approaches used should be able to be generalized to other Modelica compilers . This thesis will also provide a foundation for additional research into more sophisticated precompilation techniques for Modelica code.

1.3 Approach

This Section describes the approach used in this thesis. The goal was to precompile Modelica functions from Modelica libraries and thereby reduce the compilation scope. Two such ways were investigated in this thesis.

The first approach utilized Modelica's support for external function calls, which is described in Section 2.2.5. The idea is to transpile the Modelica code into C code that is compiled into an object file by a C compiler. This code is then called by the Modelica code using the external call functionality of Modelica. The implementation of this approach is described at Section 3.1.

The second approach used LLVM to compile the library's functions into a static library, which can be linked into an executable. Two ways of extracting functions were investigated: Analysis of the Modelica functions by the frontend and by processing models and extracting their functions. This does, in contrast to the first approach, not change the Modelica code. The implementation of this approach is described at Section 3.2

For each of the approaches, we used a benchmarking suite to evaluate the performance and the correctness of the approach. The benchmarking suite was executed on each of the approaches implemented and on the standard compiler as a baseline to evaluate performance from the perspective of:

1. Compile time performance
2. Runtime performance

The benchmarking suite is described in detail at Section 3.1.4 for the approach using external function calls and at Section 3.3 for approach using LLVM.

Chapter 2 gives a more detailed breakdown of the theory behind the thesis. Chapter 3 describes the approaches used, and the implementation process of them. Chapter 4 details the results while Chapter 5 discusses these results. Chapter 6 presents the conclusions, discussion of future work and validity threats of the thesis. In Appendix A there are boxplots of the results of the performance measurements.

Chapter 2
Background

This Chapter details the relevant theory for this thesis. More specifically, it describes compilers and the compilation process, the Modelica language, JastAdd, LLVM, and an overview of the general Modelica compiler with a description of the Modelon compiler.

2.1 Compiling

This Section will explain the basis of what a compiler does and details the different parts of a general compiler. The compiler for Modelica specifically can be seen in Section 2.6.

A compiler is a program designed to generate other programs from source code. The compiler is responsible for analyzing the code and give errors upon syntax errors, type errors or other code specific errors. It is also responsible for optimizing the code, removing unnecessary instructions, while keeping the output program behavior consistent with the input program's semantics[2]. One example of a compiler is the C compiler GCC in the GNU Compiler Collection.

A compilation is the process of transforming the input code written in human-readable text into a binary format which the computer can execute. Appel[2] breaks down this process into the following typical compiler phases:

1. Lexical analysis; parsing the code into tokens.
2. Syntax analysis, which analyzes the tokens to match the structural requirements.
3. Semantic analysis analyzes code structure, such as type analysis.
4. Intermediate code generation; an intermediate representation which simplifies the process of optimization.
5. Code optimization; remove redundant operations that are not used or their results are known at compile time.

6. Code generation; generate the machine code for the target machine.

7. Linking; link the machine code to other sources of machine code, for example static libraries.

In this thesis, "frontend" of the compiler refers to steps 1-4, and "backend" refers to steps 5 - 7. For the Modelica compiler, seen in Section 2.6, an additional step is needed between the frontend and the backend.

2.2 Modelica

Modelica is an object-oriented, equation-based programming language used for modeling and simulating complex physical systems across domains such as mechanical, electrical, thermal, and fluid systems. The version of Modelica used in this thesis is 3.4.

In Modelica, every data object is an instance of a class. There are five basic types defined in the language [3]. These types are:

- Integer
- Real
- Boolean
- String
- Enumeration types

Along with the basic types, there exist classes. The Modelica Base Class is the foundation for all of these classes. There are also exists specialized classes which build upon the Modelica Base Class but has certain restrictions and/or additional features. A few examples of these are *Records*, *Models*, and *Functions*[4]. Additionally, Modelica supports the usage of multidimensional vectors, which can contain the basic types or more complex types, such as records [5].

This thesis handles primarily Modelica functions and therefore is it relevant to note that functions can use the primitive types, aggregations of primitive types (arrays and records) and external objects (for example a C struct) as input or output. Functions will be described closer in Section 2.2.3. To better understand the specialized classes, such as functions, a description of the base class is given in Section 2.2.1.

2.2.1 Modelica Base Class

The declaration of a Modelica Base Class consists of zero to all of the following sections:

- Component declarations
- List of algorithms
- List of equations relevant to the class

Note that the component declarations could be other than the basic types; a class could contain components that are other classes, such as models or records[6]. The algorithm section of a class is where code can be executed sequentially and is also used by functions for their implementation. In contrast to this, the equations are not ordered, meaning they can be stated in any order without changing the semantics. An example of an implementation using these three sections can be seen in Listing 2.1.

2.2.2 Models

Modelica supports more complex structures to handle states and to describe the equation systems defining the system. These are called **models**. Every Model can either be a simulation model or a component model. A simulation model serves as the starting point of a simulation, for example an electrical circuit. A component model can be a part of another model, for example a resistor in a electrical circuit. Models often include components, either component models or primitive types, and equations, which describes how the different components interact with each other. When compiling a model, a global equation system is built for the entire simulation model, using the connections between the component models to combine their individual equations into the global system.

A simple equation system could be described using a model as seen in Listing 2.1.

```
model eqSystem
  input Real x; /* Component declarations */
  Real y;
  output Real z;
algorithm
  z := y * y;   /* Algorithm section */
  z := z + 1;
equation
  x + y = z;    /* The list of equations */
  x * y = z;
  z = 4;
end eqSystem;
```

Listing 2.1: Basic model example in Modelica

This equation system cannot be solved by one variable at a time since x and y are dependent on one another. Therefore, these variables must be solved simultaneously, and Modelica tools must implement solvers for these kinds of systems which can be very complex. Additionally, a model's equation system can be dependent on other component models' equation systems, making them more complex than this simple example.

2.2.3 Functions

Functions are a specialized type of the Modelica Base Class. The restrictions on functions are that they cannot contain equation lists and that the function declares any components, such as elements of basic types or elements of a component model, as either input or output. Additionally, there cannot be an internal state for functions, but they can declare local variables to be used within the algorithm section. A simple example can be seen in Listing 2.2.

```
function square
  input Real x  "Input value";
  output Real y "Output squared value";
algorithm
  y := x * x;
end square;
```

Listing 2.2: Simple function in Modelica

Partial Functions

In Modelica, classes, such as functions, can be declared with the keyword *partial*, which indicates that they cannot be instantiated [3] and are therefore similar to an abstract class in C++. The implementation of a partial class is therefore defined in the sub-classes extending the partial class.

When discussing functions, a partial function must be extended and the sub-functions will inherit the input-output signature of the partial function and they must implement the algorithm. A partial function type can be passed as a parameter in a function call assuming that the function used extends the partial function in question. [7]

The use case for this construct is, for instance, a model with different physical behaviors for different environments, such as friction. Partial functions makes it flexible to change the behavior of the model by only changing the component responsible for friction calculation, declared with the partial function type, and replacing it with suitable implementations.

2.2.4 Modelica Libraries

Modelica supports libraries, one such example is the Modelica Standard Library (MSL), which contains many pre-defined models, functions and other structures relevant for modeling. Libraries can be either read-only libraries such as MSL or proprietary libraries sold by companies or they can be editable where the developer can change the library. This thesis will only look at the effect of precompiling read-only libraries. The reason for this is that precompiling editable libraries requires an extra step in figuring out if the precompiled part has changed since the precompilation was done.

The Modelica libraries used in this thesis for testing and precompilation of functions are the following:

- Modelon Vapor cycle library [8]
- The Modelon base library [9]
- the Modelica open source Buildings library [10]

2.2.5 External and Annotation in Modelica

The keyword **external**[11] is used to make a Modelica function call an external function. This can be a C function from a source code file, or it might be a precompiled function in a

static library. If an external function is used the compiler will look for it in some default location and if a developer wants to specify where the implementation can be found they can use annotations. Annotation[12] is a general-purpose language construct that can be used to add different kinds of data to a model; in this thesis they will only be used to point to external functions. An example of a function with an external call can be seen in Listing 2.3.

```
pure function addOne
  input Integer x;
  output Integer y;
  external "C" y=c_func_adding_one(x)
    annotation(IncludeDirectory=WhereToFindFile,
    Include="#include \"one_adder.h\"");
end addOne;
```

Listing 2.3: External function in Modelica

The external function call is to a C function from the file "one_adder.h". This is one way that Modelica can call external functions in Modelica. The capability of using *external* and *annotation* to call external functions is one of the features brought by Modelica.

2.2.6 Functional Mock-up Unit

Functional Mock-up Unit (FMU) is the format the compiler outputs when compiling a model. FMU is a part of the FMI standard[13] and it is a zip file containing the necessary files for simulating the Modelica model in a simulation tool. The contents of the zip file can either be compiled binaries or source files, along with documentation such as description of the model layout and other documentation from the developer.

In this thesis, the FMUs contain one XML-file, representing the structure of the model, and a binary file with the compiled machine-code. The last step of the Modelon compiler, which is used in this thesis, is to package these files into the zip-file that is the FMU. In this report the FMU will be referred to as either the *FMU* or the *executable*.

2.3 JastAdd

JastAdd is a meta-compiler system designed to be used when creating compilers. It supports Reference Attribute Grammars (RAG) to express computations over an Abstract Syntax Tree (AST) where the nodes of the tree can reference one another [14]. It supports language constructs for working with the nodes in the AST, such as inheritance of attributes from nodes which are higher up in the AST and thus have additional information that is not present in the current node.

2.3.1 JastAdd Example

A simple example of the utility of JastAdd can be illustrated with a function declaration, which would be a node in the AST with multiple children, seen in Figure 2.1, such as:

- Head-item, a node that has a list child node with the parameters declared in the head.

- Function body, a node which has a list with the statements of the function.

The children of a function declaration can have additional children, for instance the function body will consist of different statements. The type of the return statement within the function body should match the declared type of the function, but this information is not known at the level of the return statement. JastAdd enables broadcasting of attributes from a node to all nodes below it, so for this example, the return statement could receive a broadcast with the declared type of the function from the declaration node and compare it to the local type of the statement. If not equal, then there is a type-error in the code.

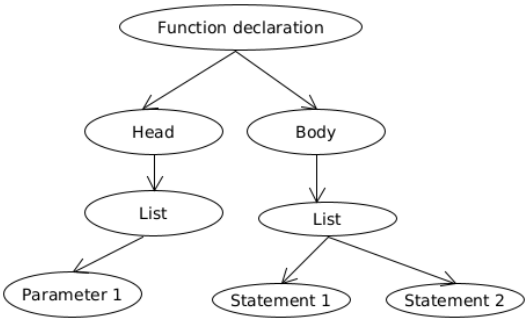


Figure 2.1: The AST for a function declaration

Thus, different types of static analysis is possible to implement in JastAdd by using RAGs and the AST structure. Static analysis is a important part of a compiler since it is extensively used in the compilation phases 3-6, which are outlined in Section 2.1.

The approach to performing static analysis with RAGs in JastAdd is one way to perform static analysis. There are other ways, but these are not relevant to this thesis.

2.3.2 JastAdd in the Modelon Compiler

In the Modelon compiler, JastAdd is used to compile Modelica into an intermediate representation, which would then be compiled into an executable. Thus, JastAdd is used for the frontend of the compiler.

In the new Modelon compiler, described more in detail in 2.6.2, the frontend outputs *Transfer-IR* as an intermediate representation. This Transfer-IR is used as an input for the LLVM-backend and is detailed in Section 2.6.2. The reason for why the Transfer-IR format exists is because the JastAdd frontend is in the process of being replaced by an LLVM-based solution.

2.4 LLVM

LLVM is a open-source compiler framework originally created by Lattner and Adve [15] to "provide a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages"[16].

The LLVM project [16] is the organization working on LLVM and LLVM is currently used as a backend to many well-known compilers such as Clang, Rustc and Swift [16] [17]. Clang is the frontend which is developed by the LLVM project.

2.4.1 LLVM in This Thesis

LLVM is relevant to this thesis since LLVM is used in the Modelon compiler as the backend to generate the executable file. For the Modelon compiler, there are no general LLVM-optimization passes done within LLVM because of time constraints when executing the compiler. In this thesis we intend to be able to use optimization passes to improve runtime performance by running the passes on the Modelica functions that are precompiled.

2.4.2 LLVM IR

To be language independent, LLVM operates on LLVM IR, which is an assembly-like language. LLVM works by letting a frontend, such as Clang, create a representation of the source-program in LLVM IR which is used to run many different levels of optimization. When the code to compile has been lowered to LLVM IR, the optimizer can work on the entire code base simultaneously, thus enabling more optimizations than could be done while compiling different modules by themselves. This is a trade-off between possibly being able to perform more optimizations but also having a longer time for compilation due to the compiler working with more data.

LLVM IR is lacking higher level language constructs, such as classes [15] and an example of LLVM IR can be seen in Listing 2.4. This example comes from emitting IR from a hello-world program, written in C, using the Clang frontend. On lines 9-14 the main method can be found with the call to printf on line 12. The declaration of printf can be found on line 16, here it is declared that there exists a function called printf which accept these parameters. It is however, not defined but expected to be defined in some other file or library.

LLVM IR is written on Static-Single-Assignment-Form (SSA-Form), which was proposed by Zadeck et al [18][19], and aims to create a version of the code which is more easily optimized by the compiler.

```
1  ; ModuleID = 'hello.c'
2  source_filename = "hello.c"
3  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128
       :128-f80:128-n8:16:32:64-S128"
4  target triple = "x86_64-pc-linux-gnu"
5
6  @.str = private unnamed_addr constant [13 x i8] c"hello world!\00", align
       1
7
8  ; Function Attrs: noinline nounwind optnone uwtable
9  define dso_local i32 @main() #0 {
10   %1 = alloca i32, align 4
11   store i32 0, ptr %1, align 4
12   %2 = call i32 (ptr, ...) @printf(ptr noundef @.str)
13   ret i32 0
```

```
14 }
15
16 declare i32 @printf(ptr noundef, ...) #1
17
18 attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"
       "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-
       protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+
       cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
19 attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
       protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+
       cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
20
21 !llvm.module.flags = !{!0, !1, !2, !3, !4}
22 !llvm.ident = !{!5}
23
24 !0 = !{i32 1, !"wchar_size", i32 4}
25 !1 = !{i32 8, !"PIC Level", i32 2}
26 !2 = !{i32 7, !"PIE Level", i32 2}
27 !3 = !{i32 7, !"uwtable", i32 2}
28 !4 = !{i32 7, !"frame-pointer", i32 2}
29 !5 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
```

Listing 2.4: LLVM IR of a hello world program

The LLVM IR is built by lowering all source code giving LLVM access to all source code. When the source code calls functions that are not defined in the source code LLVM can use "external" functions through declaration of the function names without implementing the function body for the function within the current IR file. Instead, LLVM can link libraries with the implementation of the desired functions, thus utilizing functionality that is not implemented within the IR. An example of this is the inclusion of printf in Listing 2.4.

LLVM-modules

An LLVM-module is the top-level container for all LLVM IR in the current compilation scope; it contains the list of global variables and functions this module depends on. When compiling LLVM IR, it is not uncommon to combine multiple compilation modules to one program to give the compiler more information to work with possibly enabling more optimizations. If so, there are certain restrictions which apply for the modules.

In LLVM IR, duplicate symbols are not allowed in the module scope, meaning that there cannot exist two functions, variables or other name-declared symbols with the same name within the same module. Therefore, when combining multiple LLVM IR modules, the symbol names must be unique to avoid collisions. LLVM supports declaring a variable *internal*, thus making it only visible within the current module.

2.5 Libraries and Linking

This Section describes what code libraries are, how they can be generated and what the linking of a library into an executable means.

2.5.1 Libraries

Library files are a collection of code compiled for a specific architecture, for example *x86* and in this thesis we will work in a Linux environment and thus only discuss libraries from a Linux point of view. Libraries consists of machine code ready to be relocated into the executable, without any need of compiling, since the compilation process has already been performed on the code. There are two types of libraries: static and dynamic.

Static libraries are stored in an archive file, and they consist of a number of different object files. The object files contain the compiled code, and the archive file contains a table required to find a specific library function[20]. When using a static library, the object files containing the code required by the compiled program will be linked into the executable by the linker and used for a single process; meaning multiple programs using the same library function would have multiple copies of that object file in memory.

Dynamic libraries differ from static through being designed to be shared between many different executables. This reduces the size of the executables, since they do not have a local copy of the code from the library but a reference to the shared library[20]. The use of dynamic libraries will not be explored in this thesis.

2.5.2 Linking of Static Libraries

Linking is the last step of creating a executable file. It is performed by the linker and is used when code from libraries, such as the standard library of the relevant language, is inserted into the program. The code that is linked is platform specific code, meaning it is a bit-representation of the assembly code constructed specifically for that platform (*x86*, *ARM*, *RISC-V* etc). The linker will only include the object files containing code that is used by the program, thereby reducing the size of the produced binary. [20]

2.6 Modelica Compiler

This Section details the essential parts of the compiler, describing a general Modelica compiler and the current and legacy compilers used by Modelon.

2.6.1 General Modelica Compiler

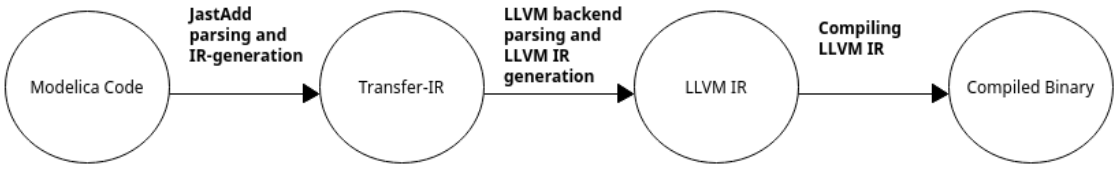
In general, the compilation of Modelica consists of the following steps[21]:

1. Frontend, which parses the Modelica files.
2. Middle end, which performs some transformations of the Modelica code such as instantiation, flattening and ordering of equations.
3. Backend, which performs code generation to create a runnable simulation file.

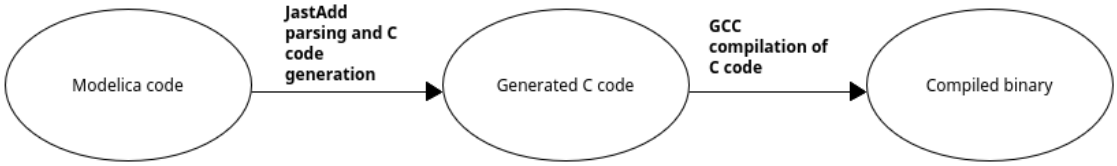
Compared to the compiler, described in Section 2.1, a Modelica compiler must create an execution order for the program. This comes from the fact that equations are declarative and it is not obvious which equation should be solved first, as described in Section 2.2.2. This is why an additional component, "Middle end", is introduced in the compiler.

2.6.2 Modelon's Modelica Compiler

Modelon is currently in the process of switching from an old compiler to a new one. The new implementation utilizes a JastAdd frontend, which transforms the Modelica code into a Transfer-IR, which is taken as input to LLVM and an executable (*FMU*) is produced. It is at the LLVM stage that linking of external libraries occurs. The compiler is visualized in Figure 2.2a.



(a) An overview of the new compiler



(b) An overview of the old compiler

Figure 2.2: Overview of the new and old compilers

Transfer-IR

When using the LLVM-based compiler, the frontend generates an IR called *Transfer-IR* and this IR contains the relevant information for the LLVM backend to generate an executable. Global variables, functions and algorithms are some examples of what is included in the Transfer-IR. Additionally, within the Transfer-IR all dependencies of the processed model are included.

The information from the Transfer-IR is used to construct the LLVM IR, which is compiled to the executable. If additional functions, which are not used by the Modelica model, are included in the Transfer-IR with their dependencies solved, then the generated LLVM IR will include these functions.

The legacy compiler, used before LLVM was introduced into Modelon Impact, generated C code as the result of the JastAdd frontend. The C code could then be compiled into an executable. The legacy compiler is visualized in the Figure 2.2b.

2.7 Precompiling Modelica

This Section details the previous work done in researching precompilation of Modelica and the difficulties in precompiling Modelica code.

2.7.1 Previous Work - Model Exchange

This Section presents the previous research for precompiling Modelica code and a technique to precompile Modelica modules.

A technique supported by Modelica to compile models and combine them is through an interface called *Model Exchange* (*ME*) which is part of the FMI-standard. It is a tool independent interface for how to combine Functional Mock-up Units (*FMUs*) [22].

ME is a system for combining compiled models, FMUs, into others thereby reducing the compilation scope for the compiler by avoiding the processing of the Modelica source code for these models. For instance, in a power plant there might be a heat exchanger and a reactor core that needs to be cooled that are modeled in Modelica. These can be compiled into FMUs and then used by a control system via the ME-interface. This control system can be implemented as a completely different program in another language. This modular approach to compiled models is similar to the goal of this thesis to precompile functions. However, there is still the requirement to solve the equations from the compiled model, since it must be done on a global scope, which does not apply to our investigation of functions since the functions are dependent on the equation system.

2.7.2 Challenges in Precompiling Modelica

Precompilng Modelica libraries is not as simple as precompiling libraries from other programming languages. This is because Modelica is a declarative language. This means that it does not matter where the equations are defined since they are all building a global equation system. This means that equations from different component models can interact and effect how the equations can be solved. This means that there is an upside to having all equations available in the compiler when doing symbolic simplifications which would not be the case if models where precompiled.

Example

In Listing 2.5, there is an example of where precompilation of a model is not possible. In the model **Gain**, there is one output, called *gain_out*, and one input, called *gain_in*. They have a simple relation with one multiplication that would seem possible to precompile. However, in **TestModel**, where **Gain** is used, the input and output are switched; *gain_in* is output and *gain_out* is input. In the compiled code this will mean that instead of an multiplication, this will lead to a division, since $\boldsymbol{y}$ will be equal to *gain_in*, which will be equivalent to $\frac{gain\_out}{k}$, thus rendering the precompiled code wrong.

```
model Gain
    parameter Real k = 1;
    input Real gain_in;
    output Real gain_out;
equation
    gain_out = k * gain_in;
end Gain;

model TestModel
    Gain gain(k = 2, gain_in = y);

    Real x = sin(time);
    Real y;
equation
    x = gain.gain_out;
end TestModel;
```

Listing 2.5: Example of a model which is impossible to precompile

Chapter 3
Method

This Chapter describes the approach used in this thesis to perform precompilation of Modelica libraries. The tools and workflows used to evaluate the implementations are also described here.

The first approach explored was the usage of external C function calls within Modelica, and it is based on the idea of modifying Modelica code to use these external calls before the compiler is executed.

The second approach uses the unmodified frontend and middle end of the compiler, but at the backend, when generating LLVM IR, we have implemented a check to see if a function is precompiled or not. This Avoids code generation of functions which are precompiled. The first approach can be seen in Section 3.1 and the second in Section 3.2.

3.1 External Function Calls

The first approach examined to reduce the scope of the compilation was to use the external function calls available in Modelica. This Modelica functionality allows Modelica functions to call C code instead of an implemented Modelica function. To achieve this, two requirements had to be met:

1. C code that can replace the Modelica functions needs to be generated.
2. The Modelica functions need to be modified to call the C function.

To accomplish this, a few changes had to be implemented in the Modelon legacy compiler, which are described in the following sections.

3.1.1 Compiler Changes

The changes we have done to the compiler were primarily to the C code generation from the Modelica code as well as an implementation of a way to create a header-file which would be

used to include the C code inside Modelica. As described in Section 2.6.2, there is functionality in the Modelon compiler to generate C code from the Modelica code. This functionality was used for our purpose of creating a static library. However, there were some difficulties in this approach that we had to address in order to get this to work.

Previously, the generated code would use C structs as parameters, and these could be used when the entire program was translated to C, but in our case, when the C functions are called from Modelica, additional constraints are placed on the parameters passed to the C functions. C structs cannot be used as parameters by Modelica since the types for the parameters in the external function call have to be Modelica types. Our approach to solving this problem was to remove the usage of C structs so that only Modelica compliant types were used in the external call from the Modelica code. However, this simplification leads to some limitations for this approach that are discussed in further detail in Section 5.1.

Additionally, we had to pass extra information about some parameters when creating the external call; more specifically the length and dimensions of vectors. This is because Modelica passes a pointer to the start of the vector, which is seen as a pointer in the C code. The vector pointer does not carry any information about the size of the vector; thus, this information had to be passed as separate arguments. We implemented this in the C code generation functionality.

3.1.2 Creating a Static Library

Using the previously mentioned changes, we created a static library by finding and isolating all Modelica functions in the JastAdd frontend and generated the equivalent C code for these. Then we used GCC to turn these C files into object files that were combined into a static library. To make the functions visible to the Modelica compiler, we had to implement an automatic generation of header files for the C functions' interfaces, then use it within the Modelica code for the include-annotations.

3.1.3 Modelica Code Modification

In the Modelica files, we had to edit the functions to call the C functions instead of the Modelica code. This was done by removing the algorithm section and replacing it with an external function call to the implemented C function. This was achieved by adding multiline comments at the start and end of the algorithm section and replacing them with the external call to the C function after the comment. Using the JastAdd frontend, we implemented an automation of this process. In the call to the function, it is necessary to insert the correct parameters needed for the C code, as explained in Section 3.1.1.

An example of how Modelica source code was changed by modification can be seen in Listings 3.1 and 3.2 with the former being the original code and the latter being the modified code.

```
within Testlib.Test1;
function F
    input Real x;
    output Real z;
    output Real y;
algorithm
    y:= x*x;
    z:= x+x;
end F;
```

Listing 3.1: Declaration of a simple function

```
within Testlib;
function F
    input Real x;
    output Real z;
    output Real y;
/*algorithm
    y:= x*x;
    z:= x+x;
end F;*/
external "C" func_Testlib_F(x, y, z)
    annotation(Library ="Testlib",
        Include = "#include <Testlib.h>",
        LibraryDirectory = "/path/to/Testlib.c",
        IncludeDirectory = "/path/to/Testlib.h");
end F;
```

Listing 3.2: Function modified to use external call

3.1.4 Benchmarking of Library Using External Function Calls

We performed benchmarking of the modified Modelica code calling external functions instead of Modelica functions by creating a benchmarking suite. The benchmarking suite consisted of a model using functions performing matrix multiplication of 16x16 matrices implemented by the authors. These matrix functions were chosen by the authors partly because all operations had to be implemented in our precompilation approach and partly because matrix multiplication is a compute-heavy operation which will show if there are some performance gains using this version of the precompilation approach. The benchmarks used for this approach of the thesis were quite limited due to the reduced functionality of the approach in general, meaning that more complex models, such as the ones detailed in Section 3.3.1, could not be used. Additional discussion of the limitations of this approach can be found in Section 5.1 and the result of the tests can be found in Section 4.1.

3.2 Libraries Generated from LLVM IR

The second approach examined for implementing precompilation of functions was to identify and replace Modelica functions directly in the compiler backend without alteration of the Modelica code. This means that the code follows the intended path of the new compiler, seen in Figure 2.2a; from Modelica code, to Transfer-IR to LLVM IR.

In this thesis, we have explored two different ways of creating the static library by using LLVM IR. The first approach, described in Section 3.2.2, starts from a Modelica model and extracts the functions within to construct a static library. We will refer to the compiled library using this approach as ***ModelLib***. The second approach, described in Section 3.2.3, performs an analysis of a Modelica library through traversing the file structure and adding functions to the frontend AST. This AST is then used in the compiler to create a static library. The compiled library using this approach will be referred to as ***ASTLib***.

3.2.1 Building the Static Library

Both approaches presented in this Section follow the same steps to create the static library but in a slightly different way. The steps to implement the precompilation of Modelica functions into a library, using the LLVM based compiler, are the following:

1. Identify functions to precompile and create a list of these.
2. Filter out functions that, for one reason or another, cannot be precompiled.
3. Generate Transfer-IR for these functions.
4. Generate the LLVM IR without model dependencies. This is detailed in Section 3.2.2 *Combining Functions from Many Models*
5. Compile the LLVM IR into a library file.

The issue with implementing these steps is that the compiler is designed to compile ***models*** and the code used by the model, not individual functions. Therefore, the compiler had to be adapted in order to use it for our purpose of compiling functions.

3.2.2 Building ModelLib

One way of solving these five steps, presented in Section 3.2.1 is to process a model in the frontend, thus generating valid Transfer-IR. The frontend uses call analysis to find all functions used in the function, meaning that the functions in the Transfer-IR might come from different libraries but they are all used by the model. When the compiler reaches the LLVM-stage of the compilation, the generated LLVM IR is filtered out to only include the functions to precompile.

Building the library from a single model and then using it for the same model yields a very high coverage of the model's functions due to the fact that all functions used by the model that can be precompiled have been processed and packaged into the library. This yields a minimum overhead in the linking and packaging process because all linked code is used by

the model; therefore, it is a best case scenario for this approach. The results of tests with the ModelLib can be seen in Figure 4.2.

The model used in testing this approach was the *Gas Extraction* model from the Modelon Modelica library *Vapor Cycles*. The Vapor Cycle library is a commercial library containing Modelica code for the development and analysis of vapor compression cycles[8]. The model itself is one of the test models used within Modelon to ensure that the compiler behaves as predicted. It was chosen since it is a model with the most calls to Modelica functions of the models investigated by the authors, as seen in Table 3.1. The model used in Section 3.1.4 was not used as the authors specifically wrote it for use with external calls due to the limited functionality of the external call approach. The impact of precompilation of the functions for the Gas Extraction model has the potential to be large on both compilation and runtime performance due to the number of functions and amount of function calls.

Combining Functions from Many Models

To create a static library with this approach from many models and not a single one, we processed multiple models and extracted their functions into object-files with one object-file representing the functions from a single model. If a model uses a function which has been precompiled earlier by another model, the function will exist in a list of precompiled functions. Meaning, a check whether a function is present in the list decides if the function is excluded from the compilation, thus avoiding duplicate symbols being defined in the library. These object-files were combined into a library. We automated the entire process of combining functions from many models and the list of precompiled functions was also implemented by us. However, combining multiple models caused issues with step 4 in the precompilation approach, detailed in Section 3.2.1; model dependencies.

When using one model for both building the library and then for testing, accesses to the surrounding context do not cause any issues due to identical structure in both the library and the model. However, if a function is being precompiled in the context of one model and used in the context of a another model, the structure of this context is different. For example, there is an array of global variables in each model containing records and variables that are used in many places in the model. This structure will be referred to as "**GlobVars**". The structure of GlobVars is different depending on what variables and records exists in that model. Thus, memory access to the same location in GlobVars would return different types depending on the execution context.

To mitigate this problem, GlobVars was transformed from being global across the entire compilation scope to being one internal array for each LLVM-module. This means that when the precompiled functions access a member of GlobVars they will instead access a local array with the correct structure and data expected by the function.

To initialize GlobVars, the compiler has a function called in the startup phase of the simulation. This function performs reading of records, constant arrays, constant variables and stores them in the global array. When using our approach, an initialization function is created for each compilation module to fill the local array with data. It is called once; the first time a function in a LLVM-module makes an access to this array.

Some functions are model specific, for example model algorithms. They are a special type of function that exists in certain models and are tied to their respective model. In the LLVM-backend they get a standard name; for example "algorithm0", since they are only intended to

exist within that module. This is problematic when we add functions from multiple models together and thus get many "algorithm0" declarations inside of the library, causing duplicate symbols which makes the library unusable. These algorithm functions have been filtered out by the authors while creating ModelLib to mitigate this conflict.

Using this implementation, ModelLib was created by extracting functions from the test models within the following libraries:

- Modelon Vapor cycle library [8]
- The Modelon base library [9]
- the Modelica open source Buildings library [10]

The test models within the Modelica libraries are designed to test the functionality of each library. ModelLib contains about 19000 precompiled functions and the full details of ModelLib can be seen in Table 4.6. The results from the benchmarking suite using ModelLib is presented in Section 4.2.

3.2.3 Building ASTLib

The second approach to create a static library using most of the existing compiler was done by not starting from a model, but instead traversing the Modelica library directly. This traversal is done by starting at the root directory in the Modelica library and going through all Modelica source code searching for functions. We add each found function to the compilation scope. Thus, all functions present in a library will be processed and not only the ones that are used by a model. We compile the functions without any context from a model, therefore, functions that are dependent on this context can not be processed using this approach.

An example of such dependencies can be seen in the package **Modelon.Media.Template** that contains several functions using properties of different mediums. A model that uses these functions is required to provide the data of the medium, for example water, thus providing some constants that are used for this medium, for example density or heat capacity. These constants are used in functions to calculate how the specified medium would react in different situations. Since the Template library does not contain the values for any material but only the functions accessing this data, these functions cannot be used in the current setting.

Another reason functions could not be precompiled using this approach was that they were deprecated functions that follow Modelica Standards that are not supported by the Modelon compiler. These functions have in general been replaced by newer functions performing the same task but are, for legacy reasons, still left in the Modelica code base.

The functions impossible to precompile were prevented from being added to the compilation scope by the use of a blacklist, thus we created a functioning library from the remaining functions. The results from the benchmarking suite using ASTLib is presented in Section 4.2.3.

3.2.4 Using the Precompiled Library

The compiler must also be modified to use the precompiled libraries, and it is broken down into the following steps:

1. Identify the functions already precompiled
2. Remove these functions from the compilation scope
3. Link the precompiled library, containing the implementations of the functions, into the compiled model.

To identify the functions already precompiled, a list containing the function names of precompiled functions was implemented by the authors. The names are unique inside the list due to the library-path being part of the name such as *Modelica.Media.Common.Helmholtz_pT*.

Removing the functions from the compilation scope is done by changing the step in the compiler where it reads the Transfer-IR of the functions and turns them into LLVM IR. If a function is found to be already precompiled instead of transferring the entire function into LLVM IR only the function header is declared within the IR. This header is needed when other functions call the precompiled function but since it is merely a function header, it does not need to be processed in the various compiler stages.

The compiler has functionality for linking libraries when compiling a model into an FMU, thus adding the library with precompiled functions is trivial.

3.3 Evaluation

This Section describes how the evaluation of the created libraries was done; both from a performance perspective and from a validation perspective.

3.3.1 Large Modelica Models

To test the performance of the precompilation strategies, multiple larger models were tested. The models chosen for the benchmarking suite were varied in their compile time, complexity and simulation time and some statistics about them can be seen in the Table 3.1. The models were selected to evaluate the approach under diverse conditions, ranging from high improvement potential to no improvement potential.

The different models were:

- Air Conditioning Example from the *Modelon Vapor Cycle library* [8]
- Data Center Continuous Time Control from the open source library *Buildings* [10]
- Data Center Discrete Time Control from the open source library *Buildings* [10]
- Frequency Sweep Force2 from the *Modelon base library* [9]
- Gas Extraction TankOnly from the *Modelon Vapor Cycle library* [8]
- XML Test Functions6 from the *Modelon base library* [9]
- VAV Reheat Guideline from the open source library *Buildings* [10]

The models come from different sample libraries, such as Modelica Buildings[10] for the Data Center models and VAV Reheat. The others are internal libraries created by Modelon. The different models' properties can be seen in Table 3.1:

Model Name	Functions	LLVM IR Size (MB)	Calls to Modelica functions
Air Conditioning	964	22.5	202 541 992
Data Center Cont	53	3.0	1 174 744
Data Center Disc	53	3.1	58 070 426
Frequency Sweep	52	17.0	8 344 762
Gas Extraction	197	12.4	449 055 617
TestFunctions6	6	154.4	6
VAV Reheat	117	28.2	126 306 838

Table 3.1: The different models' properties, including the number of function calls to Modelica functions

Running the Performance Tests

The tests for each model were executed as follows:

1. Compile the model without a library and run the executable.
2. Compile the model with ModelLib and run the executable.
3. Compile the model with ASTLib and run the executable.
4. Document the times from step 1-3.
5. Repeat step 1-4 for 100 iterations.

A run of this test takes around 24 hrs and during this time the computer was left locked with no other user program active to minimize interruptions coming from the OS doing work in the background. To reduce the variance between runs, the CPU clock frequency was set to 4.0 GHz (Maximum turbo for our CPU is 4.8GHz) through userspace CPU frequency limiting. This was done to minimize thermal throttling on the CPU and have a more consistent clock frequency.

The results of these tests can be seen in Chapter 4 and the times presented in the result is the time spent in the backend of the compiler and the time spent simulating the models.

Note that the compilation time of the precompiled libraries does not count towards the complete compile time; these libraries are already precompiled when the benchmarking begins.

3.3.2 Validity Testing

Arguably, the single most important property in a compiler is the validity of the compiled program. A compiler should not implement an optimization that changes the output [23] and it is therefore critical that our optimization does not change the output of the program. To examine the validity of the program, a validation-suite of models was used for both precompiled and non-precompiled versions and the outputs of the different compilers should yield the same result. This suite was provided by Modelon and is used for their internal validation containing hundreds of models and their expected simulation results. Thus, each version,

baseline, ModelLib and ASTLib were passed through the validation suite. The results can be seen in Table 4.4.

During the testing of the validation-suite, the performance of the compiler is not measured, only the output is evaluated compared to the baseline's result.

3.3.3 Profiling of Models

The compiler supports benchmarking for different stages of the compiler, meaning that internal time measurement of the stages are possible. Three of the test models were subjected to profiling and each were compiled 100 times with each version; baseline, ModelLib and ASTLib and the average of these 100 runs for each library is the result seen in Table 4.5. The three models profiled were:

- Gas Extraction (from *Vapor Cycle*)
- VAV Reheat (from *Buildings*)
- Frequency Sweep (from *Modelon base library*)

They were chosen by the authors due to each model coming from a different Modelica library and thus representing a broader picture of how the optimization could generalize across Modelica libraries. The various stages in the backend that were profiled were the following:

1. Setup
2. LLVM IR generation
3. Object Code generation
4. Linking of libraries
5. Building of the FMU

The expected result with both of our approaches is a reduction in LLVM IR and object code generation due to the removal of functions to process. Additionally, it is expected that the approaches will create an overhead in the linking of libraries, since more libraries are required to be included, and a longer build time of the FMU. The longer build time of FMU should be expected due to the increased data from linking of library files because the entire content of every object-file required is added into the process, even though some functions within the object file are unused. More information about creating FMUs can be read in Section 2.2.6

3.4 Test Environment

The computer setup for testing were provided by Modelon. To make the tests comparable the same hardware and software was used for all tests. The hardware consists of a four core Intel CPU, the Core-i7 1185G7, with 8 hardware threads combined with 32GB of unified DDR4 memory. The CPU was limited from userspace to run at 4.0 GHz.

The computer used Ubuntu 24.04.4 LTS as the operating system. All tests where run inside a Docker development container based on Ubuntu 22.04.5 LTS. The version of Clang

was 19.1.7, which is the same as the LLVM-version. The linker that has been used was the GNU ld 2.38. The versions of the used Modelica libraries were: Modelica Standard Library 4.1.0, Modelica Buildings Library 12.1.0, Modelon Base Library 6.0 and Modelon Vapor Cycle Library 4.0.

The version of the Modelon compiler that our solution is based on is 1.66.

Chapter 4
Results

This Chapter describes the results from our different tests and a short commentary is provided for each test result.

4.1 External Function Calls

This Section contains the results from the benchmarks conducted on the model built for usage with external function calls described in Section 3.1.4. The results, as seen in Figure 4.1, indicates that the performance of the compiler and the simulation on this matrix multiplication model is roughly equal when using the precompiled library compared to the baseline. In Table 4.1, it can be seen that the model without precompilation was around 3% faster at the frontend stage in the compiler compared to the precompiled version. On the other hand, the runtime was slightly faster, around 2% for the precompiled version. The time spent in the backend was almost identical between the two versions and the total time was essentially identical.

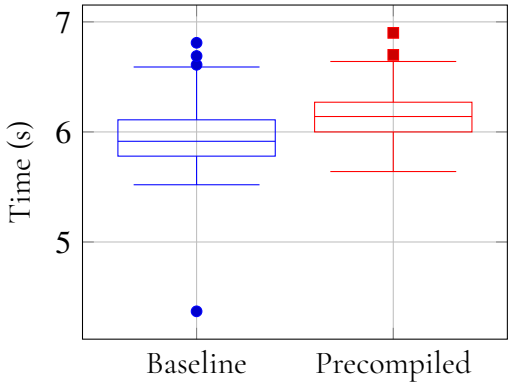
Version	Frontend time (s)	Backend time (s)	Runtime (s)	Total (s)
Baseline	5.95	53.56	2.71	62.23
Precompiled	6.14	53.32	2.65	62.12

Table 4.1: The mean time of each step in the compiler when running the matrix multiplication model

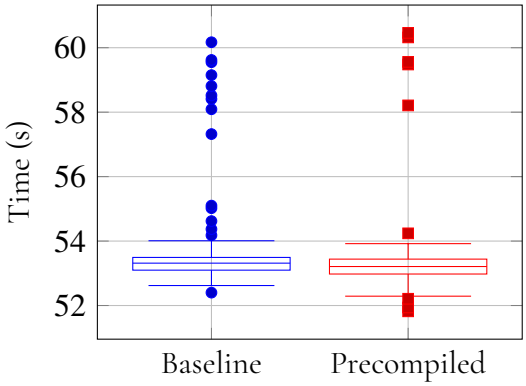
4.2 Library Generated from LLVM IR

In this Section we present the results from the tests where we have used the generated LLVM IR to create a static library that is used when compiling Modelica models. The visualization

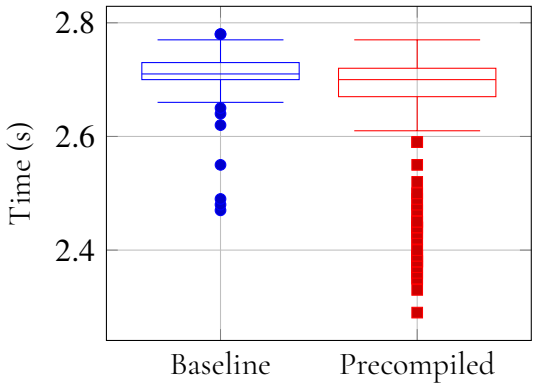
Figure 4.1: Results from replacing Modelica functions with external functions on the matrix multiplication model



(a) Frontend compilation time when running the matrix multiplication model



(b) Backend compilation time when running the matrix multiplication model



(c) Simulation time when running the matrix multiplication model

of the results can be seen in appendix A and it is presented in a table format in Table 4.3.

4.2.1 Optimal ModelLib

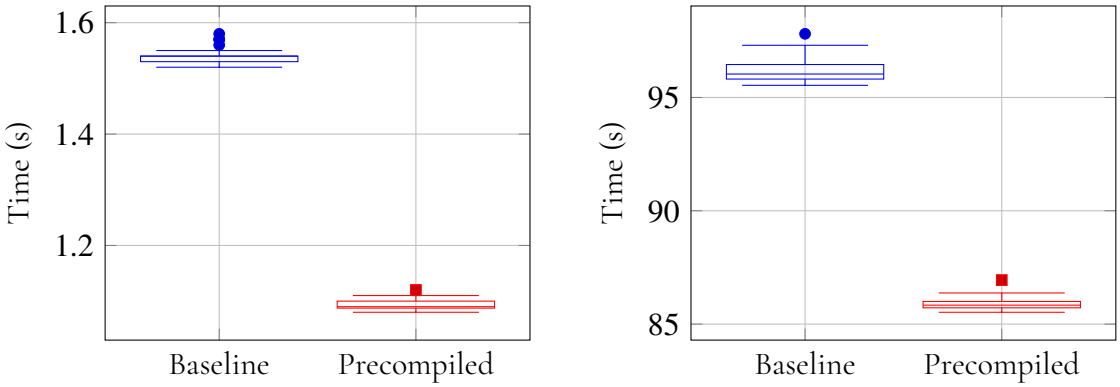
In this Subsection we will present the result from the first test using the LLVM-backend which was performed by extracting all functions from a big model and creating a static library from them. This library was used to compile the same model, meaning all functions used in the model exists precompiled in the static library.

The results from the preliminary test, using a single model for library generation and using the library on the same model, shows that both the time spent in the backend of the compiler and the time it took to simulate the model got significantly shorter when using the static library, this can be seen in Figure 4.2. For the backend compilation the results can be seen in Figure 4.2a and the results show a decrease in the time spent in compilation with

29.1%.

The runtime results, that can be seen in Figure 4.2b, show that the time spent running the simulation also decreases when using the static library. Here the decrease is 10.7 % in the average case. These results will be discussed further in Section 5.2.1.

Figure 4.2: Results from tests on the Gas Extraction model using itself as the library source.



(a) Boxplot of the time spent in the backend

(b) Boxplot of the time spent running the simulation

4.2.2 ModelLib

In this Subsection we present the results from testing of the ModelLib built from many models as described in Section 3.2.2. The results are presented in Table 4.3 and the box plots of the measurements are in Appendix A.

As can be seen in Table 4.3, there are some models that have a decrease in both simulation time and compilation time. Especially the Air Conditioning and Gas Extraction models have faster results while using ModelLib compared to the baseline. These are not as fast as the preliminary results from Section 4.2.1 for the Gas Extraction model, but they still outperform the baseline. For the other models the time saved by our approach do not exceed an introduced overhead, leading to worse performance at compile time. Discussions about possible reasons for this can be read in Section 5.2.1

4.2.3 ASTLib

This Section will present the results from the tests described in Section 3.2. The results were mixed with some models performing significantly better while being precompiled and others not being affected as much. The variance between models can be traced down to how significant part functions are of the models, with models performing better when there are more functions and function calls that benefit from precompiling. The results can be seen in a compact format in Tables 4.2 and 4.3 and the boxplots of the performance measurements are presented in appendix A.

In Table 4.2 and 4.3 it can be seen that the models with the largest reduction in IR file size after precompilation are also the ones that have the most significant change in both compilation time and simulation time. This is expected since these are the models that have been affected the most of the precompilation and thus can benefit the most from its upsides.

The results for the model TestFunctions6, which can be seen in Table 4.3 are worth noting since this model serves as a control model for what will happen when there are no functions to precompile. The model is a big model, 154.4 MB, with almost no functions to precompile which means that it shows the overhead that is introduced by our approach. The increase in compilation time for this model is roughly two tenths of a second. We deem this size of the overhead to be acceptable. We also see that the overhead is not dependent on the model size or complexity but rather on the library size. We believe this to be reasonable given that most of the overhead comes from linking and packaging the executable which will be slower when there are more object code to handle. The profiling results are presented in Section 4.2.5.

Model name	**ModelLib**		**ASTLib**	
	LLVM IR size change in MB(%)	Number of precompiled functions	LLVM IR size change in MB(%)	Number of precompiled functions
Air Conditioning	-5.80 (-27.1)	964(100 %)	-0.34 (-1.6)	101 (10.5 %)
Data Center Cont	-0.25 (-8.7)	46 (86.67 %)	-0.04 (-1.4)	17 (31.67%)
Data Center Disc	-0.26 (-8.8)	46 (86.67 %)	-0.04 (-1.4)	17 (31.67%)
Frequency Sweep	-1.20 (-7.2)	52 (100%)	-1.20 (-7.2)	52 (100%)
Gas Extraction	-3.80 (-32.0)	197 (100%)	-0.22 (-1.9)	25 (12.7%)
TestFunctions6	0.00 (0.0)	4 (66.7%)	0.00 (0.0)	4 (66.7%)
VAV Reheat	-0.92 (-3.4)	75 (63.9%)	-0.20 (0.74%)	23 (19.7%)

Table 4.2: The precompiled models' sizes compared to baseline using both extraction approaches

Model name	**ModelLib**		**ASTLib**	
	Mean backend time change in ms (%)	Mean simulation time change in ms (%)	Mean backend time change in ms (%)	Mean simulation time change in ms (%)
Air Conditioning	-503 (-20.2)	-3424 (-7.6)	471 (18.9)	-770.6 (-1.7)
Data Center Cont	105.6 (18.2)	-2.2 (-0.2)	357.7 (61.6)	-0.7 (-0.1)
Data Center Disc	72.0 (12.3)	129.5 (0.3)	318.5 (54.5)	-155.0 (-0.4)
Frequency Sweep	35.4 (2.2)	577.8 (2.1)	410.2 (25.8)	252.2 (0.9)
Gas Extraction	-310.3 (-20.5)	-8708.8 (-9.1)	426.6 (28.2)	-4522.3 (-4.7)
TestFunctions6	186.8 (3.0)	1.2 (0.2)	545.7 (8.8)	-1.3 (-0.2)
VAV Reheat	10.6 (0.4)	-588.8 (-0.5)	279.9 (10.4)	-953.4 (-0.8)

Table 4.3: Changes in time between both extraction approaches in milliseconds and percentage compared to the baseline.

4.2.4 Validity Tests

To validate that the compiler still produces the same outputs as expected when using the precompiled libraries the Modelon validity tests were run. The results from these test can be seen in Table 4.4. The optimized versions had the same success rate as the baseline compiler implementation. The reason to why the validation of the models did not reach 100% of the models tested is because the baseline compiler is not production ready, thus having certain restrictions on what Modelica models it can compile. However, since the compiler with the precompiled libraries fails the same models in the same way as the baseline we determine that it is likely that the precompilation of functions have not effected the output of the program.

Metric	Counts		
	Baseline	ModelLib	ASTLib
Tested Models	356	356	356
Successful Models	322	322	322
Compilation Errors	0	0	0
Simulation Errors	19	19	19
Verification Errors	15	15	15

Table 4.4: Results from validity test

4.2.5 Profiling of compiling

The profiling of 3 models can be seen in Table 4.5. Note that the profiling caused additional overhead in compilation time and that the numbers seen in Table 4.5 are for comparison within the table only to view where additional overhead is located.

Notable with these results are that when using the libraries, there is a large overhead in the linking and building the FMU stage compared to the baseline. This overhead is larger when using ASTLib compared to ModelLib. In Table 4.6, some stats for the different static libraries are presented and it is clear that the building of the FMU takes much more time for ASTLib compared to ModelLib.

Library Used	Backend Setup	LLVM IR	Object Code	Linking	Build FMU
Gas Extraction					
Baseline	76 (100%)	73 (100%)	999 (100%)	45 (100%)	232 (100%)
ModelLib	81 (107%)	53 (73%)	660 (67%)	55 (122%)	295 (127%)
ASTLib	78 (103%)	74 (101%)	989 (99%)	65 (144%)	624 (269%)
VAV Reheat					
Baseline	210 (100%)	154 (100%)	1638 (100%)	52 (100%)	377 (100%)
ModelLib	226 (108%)	166 (108%)	1570 (96%)	66 (127%)	513 (136%)
ASTLib	219 (104%)	156 (101%)	1624 (99%)	64 (123%)	749 (199%)
Frequency Sweep					
Baseline	95 (100%)	87 (100%)	1093 (100%)	39 (100%)	161 (100%)
ModelLib	102 (107%)	81 (93%)	1058 (97%)	50 (128%)	223 (139%)
ASTLib	98 (103%)	80 (92%)	1028 (94%)	62 (159%)	577 (358%)

Table 4.5: Profiling for some of the test models, measuring the time in **milliseconds** for the different stages in the backend and the percentage compared to the baseline.

Library	Size (MB)	Number of Functions	Number of Object Files
ASTLib	12	4058	2
ModelLib	163	19271	4357

Table 4.6: Library statistics

Chapter 5
Discussion

This Chapter discusses the findings from Chapter 4.

5.1 External Function Calls

The results from using external function calls did yield a small increase in performance and reduction in compilation time. The gains were quite small, and in the frontend, the precompiled version was slower, which is most likely caused by the compiler performing accesses to the file system to locate the files pointed to by the include annotations. Overall, the difference between the baseline and the precompiled functions using external calls were so small that no clear conclusions regarding performance can be drawn. There were, however, a few issues with this approach which were discovered once testing began.

The first problem was that the changes done in the compiler limited the scope of language constructs that could be precompiled due to more complex types, such as records, lacking a corresponding structure within the C code. This is a solvable problem, but given the time constraints we decided to not rebuild the compiler for this purpose.

Due to the lack of handling of complex types in the functions' argument, the testing was limited for this implementation since the complex simulation models, such as *Gas Extraction* used in the LLVM IR-approach, could not be tested. Therefore, the validity of this approach should be investigated further before any concrete conclusions about performance can be drawn. The root cause of the minimal testing scope is the fundamental problem that Modelica uses its own language constructs in the external function calls, which must have an equivalent in the external code, which we did not have for all constructs.

An upside of the approach is that it should be generalizable to all Modelica compilers since it changes the source code to the external calls; thereby bypassing the normal workflow for the code and becomes independent of the compiler implementation. The drawback with modifying the source code is that the generated code must be equivalent, which can be difficult to guarantee. Additionally, within the Modelica Standard Library, external calls

are prevalent for time-consuming functions, such as matrix-calculations. Using external calls is therefore a proven method for precompiling functions with the caveat that the Modelica code is written to use them in this way. Modifying existing Modelica code to use external functions in scenarios where it was not intended is possible, as shown in Section 4.1. To handle a general code base with this approach requires a dedicated Modelica to C code generator designed from the start with this as a functionality goal.

5.2 Libraries from LLVM IR

This Section discusses the results presented in Section 4.2. These results clearly indicate that precompilation of functions with optimizations is possible and could yield a very substantial performance gain, both regarding runtime and compile time. For example, the GasExtraction-model seen in Figure A.1, where there was a substantial reduction in both compile time and simulation time. However, these gains are not always generalizable to all the models. For example, the "TestFunctions6"-model, had worse compile time performance compared to the baseline and no performance increases at runtime. The big difference between these models are the number and complexity of the functions used within each.

5.2.1 ModelLib from one model

When running tests with the Gas Extraction model with the ModelLib built only from that model, the results are very positive; a 29.1% decrease in compilation time and a reduction in simulation time of 10.7%, as seen in Figure 4.2. It is also clear from the box plots that the results are statistically significant since the variance in both compilation time and simulation time is small compared to the difference between the results of the baseline and the test using the precompiled library.

One reason that this preliminary test had such a reduction in compile time is that all functions in the model have been precompiled and optimized into one object file. There is no unused code in the library, since the entire library is built from what is needed for this model, and upon linking, no code is included that will not be used by the model. This leads to minimal overhead in linking and packaging in the last stages of the compiler, thus most of the time saved in the generation of LLVM IR and object code is not removed by additional packaging or linking time. The model has many function calls, as can be seen in Table 3.1, which makes it possible to reduce the time required for running the simulation through optimization of the precompiled functions.

Comparison with ModelLib from many models

The difference in results when using a ModelLib constructed from only one model and a ModelLib from many models can be seen when comparing figures 4.2 and 4.2b with figures A.1a and A.1b for the model Gas Extraction. The smaller library is faster, both regarding compilation time and simulation time. This is in line with the results presented in Section 4.2.5 where it is stated that the overhead mostly comes from linking object files and packaging the FMU. When building the library from many models there will be multiple object files that are linked in and these will contain machine code that is not used. This unused code will result

in unnecessary overhead due to the packaging into FMU. This overhead could be avoided by structuring the object files so that no extra code is included when linking the object files. This could possibly be done by generating many object files with very few functions in each, but this would lead to an increased linking overhead. This is something in need of more investigation.

The simulation time is difficult to profile, due to a lack of tools for profiling the executed functions. To prove where the decrease in simulation time appears, such a tool is required. However, our view is that the ModelLib created from many models have introduced a slight overhead in execution time due to the management of GlobVars, see Section 3.2.2 and is the cause of the performance loss. However, more sophisticated profiling tools are required to validate this claim.

5.2.2 Performance Changes

This Subsection discusses the findings of the performance changes, both compile time and runtime, and also discusses possible explanations for these findings.

Compile Time Performance

For the models with no performance gains, a common pattern can be seen: The reduction in LLVM IR size was very low. The models with major performance gains had a large reduction in LLVM IR size, see Gas Extraction and Air Conditioning models in Table 4.3. After profiling, see results in Table 4.5 , it can be seen that our approach primarily reduced the time to generate LLVM IR and generate object code. We think it is reasonable that the functions with large reduction in LLVM IR are the ones with improved compile time performance. However, there is a small overhead caused by our approach that is dependent on how the library is constructed. As described in Section 4.2.5, this overhead comes mainly from linking the object files and packaging the executable.

For ASTLib, the overhead was increased significantly compared to both ModelLib and the baseline, which can be explained by the profiling; due to ASTLib only containing two object files, the entire library will often be included into the FMU, thus causing additional overhead when packaging. The same problem exists for ModelLib, but significantly reduced due to the more "intelligent" division of the object files within the library.

This is also supported by the fact that the FunctionTests6-model is around 200 ms slower compared to the baseline. This model did not have a notable decrease in LLVM IR size at all and had very few and very small functions. Therefore, it does not gain much from precompiling, but we think it is still subject to the overhead introduced by linking object files with more code than required, causing additional packaging costs.

We have also introduced a small overhead in the stage of generating LLVM IR due to evaluating if a function is precompiled or not. This overhead comes from the fact that we have a file with the names of all functions that have been precompiled that needs to be read into a set, which we then use to check whether a function is precompiled or not. The parsing of the file happens only once and is done in parallel with the setup of the compiler, which means that it has completed parsing the file when the set is used in the compiler. Thus, we view this overhead in the LLVM IR generation as minimal and will not discuss it further.

We think that having an overhead that is on the order of tenths of seconds is a completely acceptable prize to pay for the gains that we have in both compilation and runtime for larger models such as Gas Extraction.

In this thesis, the functions have been excluded relatively late in the compiler, namely in the backend. Excluding functions earlier in the compiler would require the headers of precompiled functions to be included in the Transfer-IR, for usage in the backend, thus requiring a large modification of the compiler. This is a conscious restriction that we have done for this approach. However, we think it is reasonable that if the exclusion of functions were to be implemented earlier in the compiler, even more time could be saved.

Runtime Performance

The runtime performance follows a similar pattern; large reduction in LLVM IR leads to a large decrease in simulation time, which is expected behavior since more code has been optimized. The performance penalty for the worst model tested was less than 2.1%, see model Frequency Sweep in Table 4.3, and we think this is explained by the small overhead introduced in every library function relying on GlobVars, causing additional time spent initializing the local data for functions at runtime. This would also explain why ASTLib has a lower overhead compared to ModelLib for this model; ASTLib has fewer initializations compared to ModelLib since an initialization happens once per included object file and occurs the first time a called function uses GlobVars. ModelLib has many more object files compared to ASTLib, as seen in Table 4.6, thus making this a possible culprit for the overhead.

This overhead is solving the issue that memory is stored globally in each LLVM-module, and the compiler is designed to only have one module; which is for the whole model being compiled. Therefore, each precompiled module requires its own data storage location that does not interfere with the globally defined storage location and to handle this, a private storage location is declared within each LLVM-module. This is then initialized when the first function dependent on this structure of the module is used; thus creating an overhead of a branch-statement in every dependent function of the module and the initialization of the required data.

When using the compiler without our optimization, all data initialization happens once at the start of the program. In the precompiled implementation, every module initializes the data required for that specific module upon the first call to a function relying on GlobVars. This creates the possibility of duplicate data being loaded into memory for, since each module has its own memory management, causing an increase of memory usage, and a lower runtime performance due to the additional memory operations generated by having duplicated data in memory.

There is a possibility of making this problem smaller by being smarter when choosing what functions are placed in which modules. Placing functions using the same global constants in the same compilation module would make the compiler spend less time initializing data. The extreme is to put all functions in the same compilation module. One problem with this is that, if using static libraries, it will make the FMU bigger in size and more time will be spent packaging the FMU in the backend. This can be seen with ASTLib in Table 4.5.

The performance gains for the best models were up to almost 10%, as seen in Fig A.1b and A.2b, which indicates that the optimized functions are reducing the simulation time. This is the expected behavior from our optimization on the runtime performance.

5.2.3 Overhead when Using ASTLib

When using the library built through analyzing the library there is some overhead compared to both the baseline and the ModelLib. Through profiling of the compilation, a single step was responsible for the increased overhead; packaging the compiled model into the FMU format. This can be seen in Table 4.5. The ASTLib consists of only two large object files, thus it is very likely that the entire library is linked into the executable even if only a few functions are being used. This means that the executable will contain more object code than necessary which will slow down the packaging of the files. This problem could possibly be solved by splitting the library into smaller object files or using a dynamic library instead of a static one so that not all of the functions needs to be packaged together with the FMU. However, additional investigation into splitting the library into several object files actually increased packaging time even more. These preliminary tests where done in a crude way meaning there where many unnecessary declarations of function headers in the LLVM IR. All the functions were declared in each file but only a few implemented, thus increasing size of the library by more than an order of magnitude; to 170 MB from 12 MB, and the functions were divided randomly. Adding an analysis pass where the functions are divided according to what functions will tend to be used together and also not including unnecessary header declarations in the LLVM IR will possibly make this overhead much smaller. In ModelLib, this split was done because each model created an object file with the functions within that model and for this approach the packaging overhead was not as extensive.

5.2.4 Comparison and Potential Usage

The two approaches presented in this thesis serve different purposes. They can both serve a purpose in a Modelica tool but in different settings. Firstly, ASTLib can easily be incorporated in the normal release cycle of Modelica libraries. It would mean that, when releasing a new version of the Modelica library, a static library is built and bundled with it. This ensures that all functions intended for use outside the library are precompiled and will be run with optimizations that will enable faster simulation times.

In contrast, the approach extracting functions from a model can be incorporated more directly in the compiler used by developers. It can leverage the fact that developers tend to compile the same model many times during development and between these recompilations, most of the code remains unchanged. To implement this, some kind of check would need to be done to see if a Modelica function has been altered or if it is safe to use the precompiled version. This could potentially be implemented as a worker in Modelica tools, always working in the background to produce the most up to date library possible. It could also be implemented as an extra pass in the compiler trading a small increase in compile time during the first compilation of the model against creating a reusable static library that can be used during subsequent compilations. This use case is similar to the tests with the optimal ModelLib discussed in Sections 4.2.1 and 5.2.1.

5.2.5 What Functions Cannot Be Precompiled

For the approach building libraries from models, all functions, except the model algorithm, could be precompiled due to all the necessary data being available at compile time. Algorithm

functions were not precompiled due to naming collisions; the name within the library must be unique. Additionally, if the algorithms are from read-only models, there might be value in precompiling these as well but that requires solving the naming collision issue.

When building the ASTLib there were some functions that were not precompilable. The most common reason for this was missing information which is supposed to be specified by the model using the function. This information was mainly material properties, where values such as density, conductivity and other material properties were defined.

There were also some functions where it was not possible to precompile the code due to incompatibility between the Modelica code and the Modelon compiler. These were mostly legacy functions, thus the new compiler no longer supported these functions. Additionally, there were some functions using tool specific functions from an another Modelica tool, these functions are not standard Modelica and are thus out of scope for this thesis.

5.2.6 Validity of Precompilation Results

From Table 4.4, the validation results are on par with the baseline compiler implementation. The expected behavior is identical behavior and is fulfilled by the optimized versions. Thus, indicating that the compiler behaves in the same way for the different validation models.

Chapter 6
Conclusions

This Chapter summarizes the conclusions of this thesis, answers the research questions stated in 1.1 and discusses potential future work.

6.1 Research Questions

This Section answers the research questions and draws conclusions around them based on the discussions in Chapter 5.

6.1.1 Extraction of Modelica Functions

Our first research question detailed how Modelica functions could be extracted and precompiled:
"*How can Modelica functions be extracted and precompiled?*"

We found three ways of achieving this:

1. Generate C code that implements the function, compile it and replace the Modelica function with an external call to this function.

2. Process the model as usual in the compiler, but when generating the intermediate representation (LLVM IR), only include the functions used by the model.

3. Filter functions from a library in the frontend; only process the AST-nodes that are functions and generate code for these.

Of these three approaches, only the second and third yielded any results that differentiated themselves from the baseline. Therefore, for the rest of the research questions, only these two approaches will be discussed.

6.1.2 Precompilation Performance Changes

Our second research question tackled the performance side of things:
"*How does precompilation of functions affect runtime and compile time performance?*"

A clear conclusion here is that some models gain performance from precompilation of functions, while others do not. The best measurement of performance improvement for our approach, both for compile time and runtime, is the reduction in LLVM IR. A large reduction of LLVM IR yields a compile- and runtime performance increase.

6.1.3 Limitations of Precompiling Modelica Functions

Our last research question detailed when Modelica functions are unable to be precompiled:
"*When is it not possible to precompile a Modelica function?*"

If the Modelica function is dependent on information not known at compile time, then it cannot be precompiled in the context of this thesis. Such functions can be encountered in Modelica; template functions that require a model to enter media properties, such as the density of the material, in an array which is then used by the function. Without the model context, compilation of such functions are not possible to do in a efficient way, but with the model context they can be compiled as well. These functions will probably be possible to precompile by turning them into closures but we have deemed this outside the scope of this thesis.

Another limitation is the risk of name collisions between the static library and the model using the library. Such examples were the algorithm functions. These are not impossible to precompile, if relevant data is known at compile time, but there must be a system designed to avoid naming collisions. The library functions used in this thesis were named after its unique path in the library to solve this issue.

In conclusion, a Modelica function can be precompiled, just like any other language, when all dependencies are known at compile time. Additionally, naming collisions must be avoided in order to enable precompilation of Modelica functions.

6.2 Future Work

This sections discusses some potential optimizations that were not done in this thesis such as precompilation of model algorithms, a general method for optimizing dynamic functions, and exclusion of precompiled functions earlier in the compiler. These will be discussed in the context of the second approach tested with the LLVM-based compiler.

6.2.1 Precompilation of Dynamic Code

To widen the scope of precompilation, reusing code which does not belong to read-only libraries could yield additional performance gains. This would require a way to identify if the precompiled version is outdated with the current source code; if true, then compile the function as normal and replace it in the library. It would essentially be a cache for these functions, meaning subsequent compilations without changes to the dynamic functions should be faster.

6.2.2 Precompilation of Algorithms

The implementation of optimizing Modelica functions is currently excluding the algorithm functions of models. This is due to the naming convention of algorithms in LLVM IR is only a counter for them; algorithm0, algorithm1 etc, as of the time of writing. Thus, precompiling these does not yield unique function names and would cause collisions when the library algorithms and the algorithms used in the model would be merged together. Therefore, creating unique names for the algorithms should be sufficient for precompiling, if the models' algorithms are static. If the algorithms could be modified, then a method of comparing the precompiled version with the source code version is required.

6.2.3 Exclusion of Precompiled Functions Early in the Compiler

In the implementation described in this thesis, the exclusion of functions appear late in the compiler; before LLVM IR generation. This means that the frontend and middle-end have to process these functions which is additional work that should not be required to do. It would not be possible to exclude the functions completely since the function signature is needed to perform error checking. However, the algorithm section could be excluded leading to less work for the compiler.

6.2.4 Static Libraries Compared to Dynamic Libraries

In this thesis we have only created static libraries when precompiling Modelica functions. The alternative to this would be to use dynamic libraries instead. This would possibly reduce compilation time since the linker does not have to link the object files from the dynamic library into the FMU, only providing references to these functions. This should make the compiled FMU smaller, but dependent on the runtime environment, making it less portable.

A smaller FMU should cause a reduction in the time to package everything and thereby reducing the compile time of the model. However, this would possibly have a negative effect on the simulation time since the runtime environment would have to load the used functions. It might be interesting to do some measurements to decide which of these paths is better.

6.3 Threats to Validity

This thesis has explored how precompiling Modelica functions effects the compile time and runtime performance. The results presented in Chapter 4 point to there being a improvement in both compile time and runtime performance under some circumstances but there are still uncertainties that might effect the correctness of these results.

The validity tests used in this thesis to validate that the compiler produces a correct program are not completely reliable. One problem with the validity tests are that, since the Modelon compiler is still under development, not all validity tests pass for the baseline compiler. Although we have run the validation tests and the same test models pass compared with the baseline, it is possible that some of the failed tests fails in a different way for the

approach presented in this thesis compared to the baseline. These validation tests might not cover all possible Modelica functions, indicating that there could be some edge cases this approach fails. But, we view this as unlikely due to identical output compared to the baseline for the passing models.

All tests during this thesis has been performed using the Modelon compiler and the Modelon Modelica tool. This means that the thesis utilizes how the Modelon compiler represents functions internally. Another compiler with a different internal structure might get different results. However, it seems likely to us that the general findings will generalize to other Modelica tools and compilers.

A last threat to the validity of the tests are the fact that this thesis does not use section-based linking. Section-based linking is a tool built into compilers that allows for only bringing in the used symbols when linking to a static library. If this was used then the overhead caused by linking unnecessary functions into the executable would probably be removed. This would mainly benefit the compile time performance of ASTLib where there was a big performance penalty for including too much from the object files; possibly all 4058 functions. Adding section-based linking would probably not effect the conclusions of this thesis in a negative way but would rather serve to solidify the usefulness of precompiled library functions. If tests with section-based linking where performed then it is probable that the answer to RQ 2, declared in Section 1.1 would change.

References

[1] Modelica Association. Modelica tools. https://modelica.org/tools/, 2026. Accessed: 2026-02-17.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK, 2 edition, 2002.

[3] Modelica Association. Classes, predefined types, and declarations, 2025. Accessed: 2026-01-29.

[4] Modelica Association. Modelica – a unified object-oriented language for systems modeling: Specialized classes (section 4.6), April 2017. Section 4.6, "Specialized Classes".

[5] Modelica Association. Modelica – a unified object-oriented language for systems modeling: Array declarations (section 10.1), April 2017. Accessed: 2026-02-16.

[6] Modelica Association. Classes for Reuse of Modeling Knowledge. Section in Modelica Language Tutorial, GitHub repository, 2026. Basic Language Elements subsection, "Classes for Reuse of Modeling Knowledge".

[7] Modelica Association. Modelica language specification version 3.6: Functions. https://specification.modelica.org/maint/3.6/functions.html, 2020. Accessed: 2026-01-30.

[8] Modelon, Inc. Vapor Cycle Library. https://modelon.com/library/vapor-cycle-library/, 2026. Accessed: 2026-02-25.

[9] Modelon, Inc. Modelon Library Suite. https://modelon.com/modelon-library-suite-modelica-libraries/, 2026. Accessed: 2026-04-15.

[10] Lawrence Berkeley National Laboratory. Modelica buildings library. https://simulationresearch.lbl.gov/modelica/, n.d. Accessed: 2026-03-24.

[11] Michael M. Tiller. External functions. https://mbe.modelica.university/behavior/functions/external/, 2014. Accessed: 2026-02-17.

[12] Modelica Association. Modelica language specification: Annotations. `https://specification.modelica.org/master/annotations.html`, 2024. Accessed: 2026-02-16.

[13] Modelica Association Project FMI. Functional Mock-up Interface Specification 3.0: FMU Distribution, 2023. Accessed: 2026-04-23.

[14] JastAdd Development Team. Jastadd — a meta-compilation system supporting reference attribute grammars. Online, 2025. Accessed: 2026-02-13.

[15] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE Computer Society, 2004.

[16] LLVM Project. LLVM: The LLVM Compiler Infrastructure Project. `https://llvm.org/`, 2026. Accessed: 2026-02-09; official overview of the LLVM compiler infrastructure project.

[17] Swift Project. Swift Compiler Documentation. `https://www.swift.org/documentation/swift-compiler/`, 2026. Accessed: 2026-02-09; details Swift compilation pipeline including LLVM IR generation as part of code-generation backend.

[18] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, page 70–85, New York, NY, USA, 1986. Association for Computing Machinery.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 25–35, New York, NY, USA, 1989. Association for Computing Machinery.

[20] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, Hoboken, NJ, 10 edition, 2018.

[21] Dirk Zimmer. Module-preserving compilation of modelica models. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference*, volume 43 of *Linköping Electronic Conference Proceedings*, pages 880–889, Como, Italy, September 20–22 2009. Linköping Electronic Conference Proceedings.

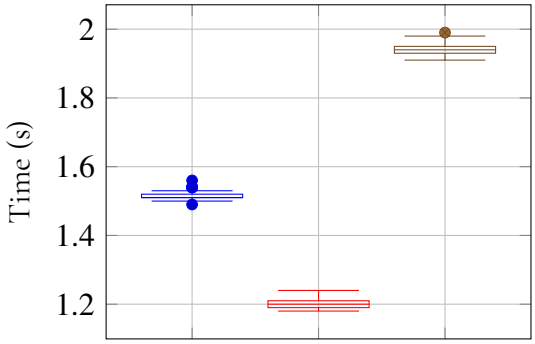
[22] Andreas Junghanns, Torsten Blochwitz, Christian Bertsch, Torsten Sommer, Karl Wernersson, Andreas Pillekeit, Irina Zacharias, Matthias Blesken, Pierre R. Mai, Klaus Schuch, Christian Schulze, Cláudio Gomes, and Masoud Najafi. The functional mock-up interface 3.0 – new features enabling new applications. In *Proceedings of the 14th International Modelica Conference*, pages 17–26. Modelica Association and Linköping University Electronic Press, 2021.

[23] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. In *Proceedings of Higher-Order and Symbolic Computation*. Springer / ACM, 2004. Proves that transformations preserve program semantics (same observable behavior).

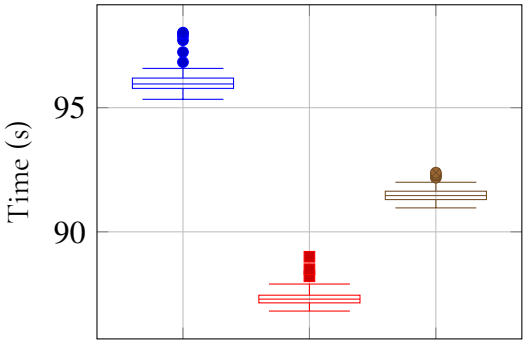
Appendices

Appendix A
Boxplots from Testing

Figure A.1: Results from the tests on the Gas extraction model.

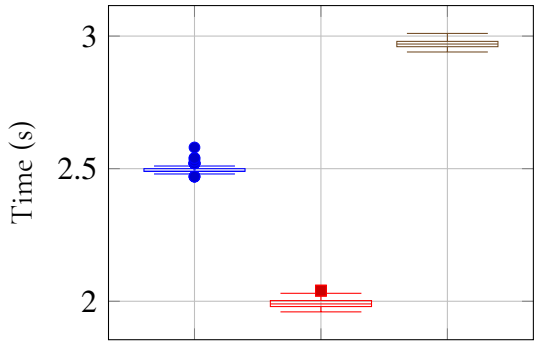


(a) Boxplot of the time spent in the backend

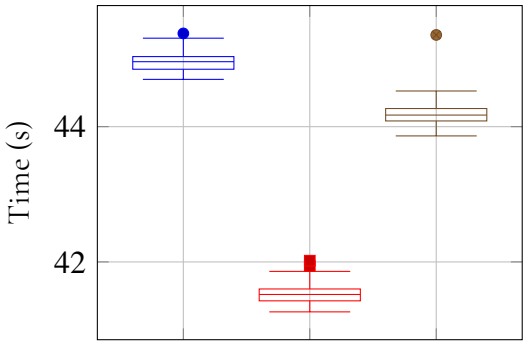


(b) Boxplot of the time spent running the simulation

Figure A.2: Results from the tests on the Air Conditioning model.

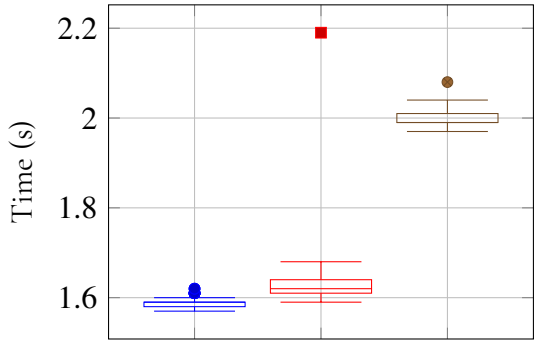


(a) Boxplot of the time spent in the backend

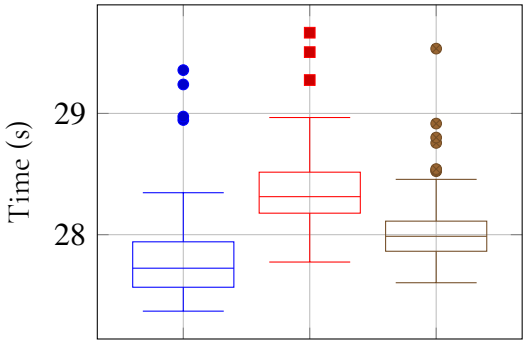


(b) Boxplot of the time spent running the simulation

Figure A.3: Results from the tests on the Frequenzy Sweep model.

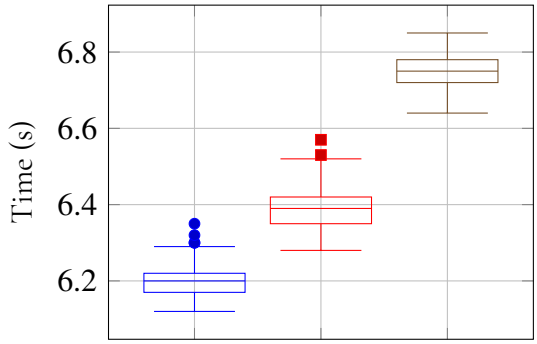


(a) Boxplot of the time spent in the backend

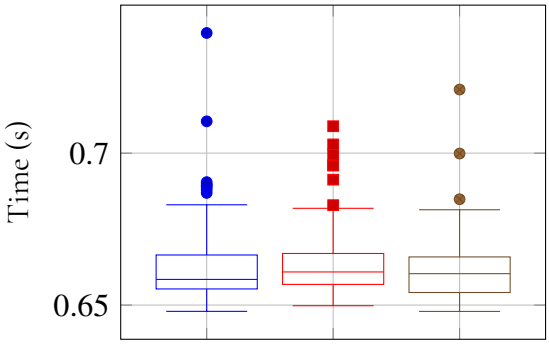


(b) Boxplot of the time spent running the simulation

Figure A.4: Results from the tests on the FunctionTest6 model.

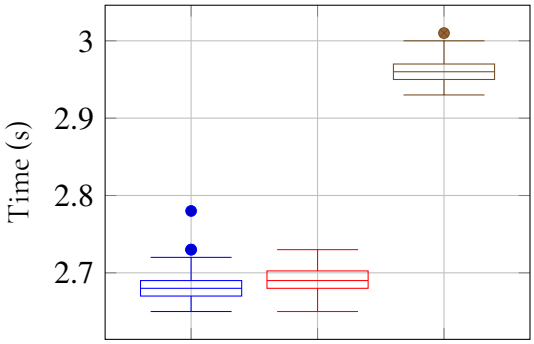


(a) Boxplot of the time spent in the backend

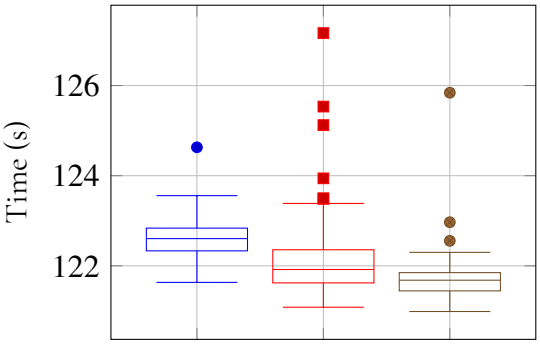


(b) Boxplot of the time spent running the simulation

Figure A.5: Results from the tests on the VAV Reheat model.

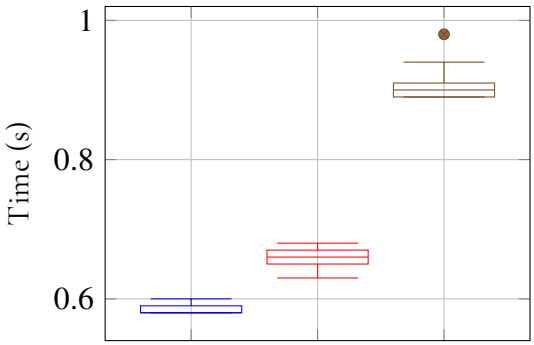


(a) Boxplot of the time spent in the backend

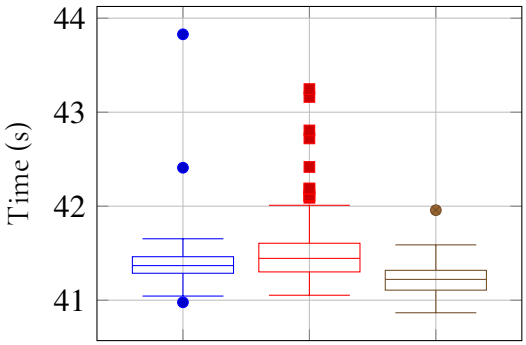


(b) Boxplot of the time spent running the simulation

Figure A.6: Results from the tests on the Discrete Time Control model.

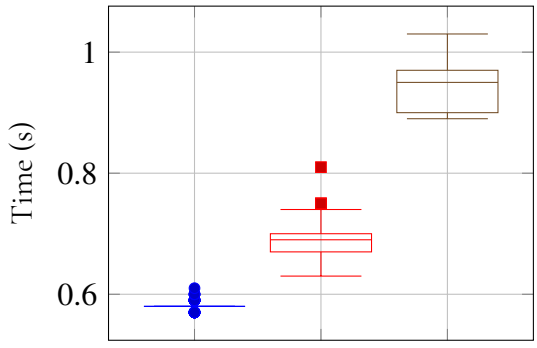


(a) Boxplot of the time spent in the backend

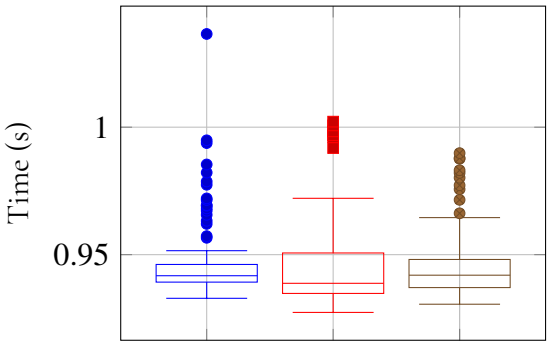


(b) Boxplot of the time spent running the simulation

Figure A.7: Results from the tests on the Continuous Time Control model.



(a) Boxplot of the time spent in the backend



(b) Boxplot of the time spent running the simulation

DEPARTMENT OF COMPUTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTED 2026-05-25

Msc. THESIS Reducing compilation scope in an LLVM-based Modelica Compiler by precompiling libraries
STUDENTS Axel Nilsson, David Lidholm
SUPERVISORS Jonas Skeppstedt (LTH), Filip Stenström (Modelon), Markus Olsson (Modelon)
EXAMINER Christoph Reichenbach (LTH)

How to reuse work we have already done

NON-TECHNICAL SUMMARY **Axel Nilsson, David Lidholm**

When we do work, it's usually better if we can finish it faster. Imagine drawing a house that has a beautiful window. If you want to add more windows, it would be easier to copy the one that is already drawn. This is a good way to save time.

In this thesis, instead of drawing windows and houses, the work is to simulate different kinds of things, for example a race car. This is done so that companies do not have to build many different versions of a car. Instead, they can test many versions on a computer.

Modelon is a company that creates a tool that can do this simulation. But this tool also takes some time to run, time that race car manufacturers want to spend on other things instead of waiting for a computer to work. The race-car manufacturer creates a model of their car in a programming language called Modelica. Before the model can be simulated, it must be turned into code that the computer understands. This process is called **compiling**. This thesis focuses on reusing parts of this compiled code.

Using old work is good but it is not always easy to do it in a good way. If every house in a village looked exactly the same, it would not look realistic. In the same way, we must be careful to only reuse parts that do not change the final result. It is not possible to reuse all of the code for the machine, so this thesis looks at which parts can be reused safely.

An artist might decide from the beginning that they are going to draw one window and then reuse it, then it makes sense to spend more time making sure that that window looks really good so that all windows in the final drawing will be stunning. This same concept applies to the work done in this thesis, the parts we know will be used many times can be improved more, so they run as fast as possible and reduce waiting time.

We tested different approaches and found that it is possible to reuse previous work in a way that reduces both the compilation time and the simulation time. How much time is saved depends on which model is being simulated.