Institutionen för datavetenskap
Department of Computer and Information Science

Master’s Thesis

Incremental Compilation and Dynamic Loading of Functions in OpenModelica

Kim Jansson, Joel Klinghed

Reg Nr: LIU-IDAl/LITH-EX-A--08/029--SE
Linköping 2008
Master’s Thesis

Incremental Compilation and Dynamic Loading of Functions in OpenModelica

Kim Jansson, Joel Klinghed

Reg Nr: LIU-IDA/LITH-EX-A--08/029--SE
Linköping 2008

Supervisor: Peter Fritzson
IDA, Linköpings universitet
Adrian Pop
IDA, Linköpings universitet

Examiner: Peter Fritzson
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
Incremental Compilation and Dynamic Loading of Functions in OpenModelica

Kim Jansson, Joel Klinghed

Advanced development environments are essential for efficient realization of complex industrial products. Powerful equation-based object-oriented (EOO) languages such as Modelica are successfully used for modeling and virtual prototyping complex physical systems and components. The Modelica language enables engineers to build large, sophisticated and complex models. Modelica environments should scale up and be able to handle these large models. This thesis addresses the scalability of Modelica tools by employing incremental compilation and dynamic loading. The design, implementation and evaluation of this approach is presented. OpenModelica is an open-source Modelica environment developed at PELAB in which we have implemented our strategy for incremental compilation and dynamic loading of functions. We have tested the performance of these strategies in a number of different scenarios in order to see how much of an impact they have on the compilation and execution time.

Our solution contains an overhead of one or two hash calls during runtime as it uses dynamic hashes instead of static arrays.

Dynamic loading, Optimization, Incremental Compilation, Compiler Construction
Abstract

Advanced development environments are essential for efficient realization of complex industrial products. Powerful equation-based object-oriented (EOO) languages such as Modelica are successfully used for modeling and virtual prototyping complex physical systems and components. The Modelica language enables engineers to build large, sophisticated and complex models. Modelica environments should scale up and be able to handle these large models. This thesis addresses the scalability of Modelica tools by employing incremental compilation and dynamic loading. The design, implementation and evaluation of this approach is presented. OpenModelica is an open-source Modelica environment developed at PELAB in which we have implemented our strategy for incremental compilation and dynamic loading of functions. We have tested the performance of these strategies in a number of different scenarios in order to see how much of an impact they have on the compilation and execution time.

Our solution contains an overhead of one or two hash calls during runtime as it uses dynamic hashes instead of static arrays.
Acknowledgments

We would like to thank our examiner Peter Fritzson and our supervisor Adrian Pop for their input and support during the work on this thesis.
Contents

1 Introduction 3
  1.1 Intended Audience ....................... 3
  1.2 Background ........................... 3
  1.3 Purpose ................................ 4
  1.4 Goals .................................. 4
  1.5 Method ................................ 4
  1.6 Limitations ............................ 4
  1.7 Thesis Outline ......................... 5

I Theoretical Background 7

2 Techniques 9
  2.1 Dynamic Libraries ....................... 9
  2.2 Virtual Function ........................ 10
  2.3 Incremental Compilation ................. 10

3 Equation-Based Object-Oriented Languages 11
  3.1 Modelica Language ...................... 11
  3.2 OpenModelica Environment ............... 12
    3.2.1 OpenModelica Modules .............. 12
  3.3 MetaModelica and RML .................. 13

II Incremental Compilation and Dynamic Loading of Functions 15

4 Dynamic Loading of Functions 17
  4.1 Variable Passing ....................... 18
## Contents

10.2.2 Static Array Function Lookups: Implementation 61

### Bibliography

### A Interfaces

- A.1 base_array Interface 65
- A.2 read_write Interface 70
- A.3 dynlib Interface 77
- A.4 strhash Interface 78

### B Performance Tests

- B.1 Process Calling Overhead 81
  - B.1.1 Result 81
  - B.1.2 Procedure 82
- B.2 External Function-call 83
  - B.2.1 Result 83
  - B.2.2 Procedure 83
- B.3 Incremental Compilation: Single Package 85
  - B.3.1 Result 85
  - B.3.2 Procedure 85
- B.4 Incremental Compilation: Multiple Packages 91
  - B.4.1 Result 91
  - B.4.2 Procedure 91
- B.5 Incremental Compilation: Simulations 96
  - B.5.1 Result 96
  - B.5.2 Procedure 96
List of Figures

3.1 Modules in the OpenModelica Compiler .................. 12
5.1 Associative Arrays ...................................... 27
5.2 Sequence for looking up functions ........................ 27
5.3 Overloading of packages and their functions .......... 28
B.1 Benchmark of a single package with nested functions on Linux ........................................ 88
B.2 Benchmark of a single package with nested functions on Windows ...................................... 88
B.3 Benchmark of a single package using many functions on Linux ............................................ 89
B.4 Benchmark of a single package using many functions on Windows ...................................... 89
B.5 Benchmark of a single package using few functions on Linux ............................................. 90
B.6 Benchmark of a single package using few functions on Windows ....................................... 90
B.7 Benchmark of multiple packages using 50% of the functions on Linux ...................................... 93
B.8 Benchmark of multiple packages using 50% of the functions on Windows ...................................... 93
B.9 Benchmark of multiple packages using 90% of the functions on Linux ...................................... 94
B.10 Benchmark of multiple packages using 90% of the functions on Windows ...................................... 94
B.11 Benchmark of multiple packages using 90% of the functions with fewer OpenModelica restarts on Linux ........................................ 95
B.12 Benchmark of multiple packages using 90% of the functions with fewer OpenModelica restarts on Windows ...................................... 95
B.13 Benchmark of simulation using 50% of the functions on Linux ............................................. 97
B.14 Benchmark of simulation using 50% of the functions on Windows ....................................... 97
B.15 Benchmark of simulation using 90% of the functions on Linux ............................................. 98
B.16 Benchmark of simulation using 90% of the functions on Windows ............................................ 98

List of Tables

B.1 Measurements of process calling overhead ................. 81
B.2 Measurements of external function calls .................. 83

Terms and Definitions

- \textit{dll} : Dynamically loaded library.
- \textit{CORBA} : Common Object Request Broker Architecture.
- \textit{External function} : a non-built-in function called by OpenModelica. Not to be confused with \textit{Imported function}.
- \textit{Imported function} : an externally implemented function in either C or Fortran referenced in Modelica using the “external” keyword. Imported functions are a subset of \textit{External functions}.
- \textit{OMC} : OpenModelica Interactive Compiler.
- \textit{OMShell} : OpenModelica Shell.
- \textit{Package function} : an external function defined in a package class.
- \textit{RML} : Relational Meta-Language.
Chapter 1

Introduction

In this chapter we will give a brief introduction of this thesis, its outline and goals.

1.1 Intended Audience

This thesis is intended for anyone interested in more technical aspects of Modelica and OpenModelica. The reader may be an OpenModelica developer or a person interested in compiler construction.

1.2 Background

Modeling and simulation of large and complex systems like cars or space shuttles is today a growing field. Thus the need for good modeling and simulation tools is apparent. The Modelica language is an object-oriented, multi-domain modeling language specialized for simulations. OpenModelica is a project with the goal to build a complete modeling and simulation environment using the Modelica language.

OpenModelica has some issues. One of them is a long turn-around time after small code changes when developing large Modelica projects. The main problem is that each changed file needs to be recompiled and the complete project re-linked each time. With sufficiently large files this takes a while. One proposed solution is to use Incremental Compilation to decrease the amount of code that needs to be recompiled.
Another issue is the use of an ineffective and inexact method of transferring variables to and from functions. This is because text files are used as medium.

1.3 Purpose

The purpose of this project is to make OpenModelica more effective when functions are called and when they are recompiled. The thesis investigates the possibility of using incremental compilation to lower turn-around time in large OpenModelica projects. It also investigates if there exists a more effective solution for transferring variables to and from functions.

1.4 Goals

The goals of this project are to implement incremental compilation and dynamic loading of functions in OpenModelica. As a side goal we will also thoroughly re-factor argument passing to and from functions. The aim is to implement these features in such a manner that developers using OpenModelica can benefit from these features without modifying their code.

1.5 Method

This project was performed at PELAB, Linköping University. As the project emphasizes on implementation there was not a great need for prestudies. This gave us the opportunity to use an agile development process in which we could implement small precise features that together formed the complete solution. Since agile development relies heavily on testing it was a big help that there already existed a large collection of unit tests for OpenModelica.

1.6 Limitations

Since the project is limited in time it was decided that our solution did not need to include a way to automatically mark modified functions.
1.7 Thesis Outline

We did not have access to any sufficiently large Modelica projects during the development of this project to test the performance thoroughly. This was limiting because the benefits of incremental compilation becomes more prominent the larger the files and the project are. When compiling small projects it may even be a disadvantage because of the extra work.

1.7 Thesis Outline

This thesis is divided into three distinct parts. The first part gives the reader the needed theoretical background, mainly focusing on three of the languages involved in OpenModelica – Modelica, MetaModelica, RML – and some of the techniques that will be used for the implementation. The second part describes the project more detailed, including the design of our implementation as well as technical details. The third part describes the result of the project – for example performance benchmarks – and conclusions drawn from them.
Part I

Theoretical Background
Chapter 2

Techniques

This chapter mainly contains a short explanation of the different techniques we used in this project.

2.1 Dynamic Libraries

A long time ago there were only static libraries; collections of precompiled code ready to be inserted into programs at compile-time. Static libraries leads to a lot of code duplication, for example all programs written in C would contain its own copy of the not so small standard C library. As a countermeasure dynamic libraries were introduced. Dynamic libraries enables programs to share code by loading these shared libraries dynamically at runtime. Dynamic libraries were not available to the programmer at first and thus they were only used by the operating system and the linker. Later on, APIs were developed to expose this functionality. [3]

Dynamic loading of libraries is to this day and age well supported in most operating systems. They are slightly different in some of the capabilities and usability but the important features are there.

For some reason most programming environments still has – if any – quite rudimentary support for dynamic loading of libraries and functions therein. For example C and C++ has no standardized support for dynamic loading even though it is a common part of most setups. Most Unix variants uses the POSIX API and on Microsoft Windows it is part of the Windows API. [7, 1]
2.2 Virtual Function

Virtual function is a term from the object-oriented paradigm. A virtual function is a function that can be redefined further down the inheritance chain. This makes it possible to call a generic BaseObject but still use the redefined function in the SubObject. In C++ this is often implemented using a lookup-table of function pointers hidden by the language syntax. We choose to use the term Virtual function even though we do not use objects as such, instead of an inheritance chain our solution uses a patch chain.

2.3 Incremental Compilation

Incremental compilation means that when the compiler compiles a project it only compiles those parts of the code that were actually modified, using some container such as block, function, class or file as lowest denominator. For example most modern compilers can compile each file as a single unit and then only recompile those that have been modified. The point of incremental compilation is to decrease the turn-around time for a recompilation, especially when only minor modifications have been made to the code.
Chapter 3
Equation-Based Object-Oriented Languages

This chapter provides a short overview of what the Modelica language is. It also contains an overview of OpenModelica - in which we will add dynamic loading and incremental compilation - and a brief introduction to MetaModelica and RML which are used to compile OpenModelica.

3.1 Modelica Language

Modelica is an object-oriented modeling language for declarative equation-based mathematical modeling of large and heterogeneous physical systems. Modelica was designed with the main objective of facilitating exchange of models, model libraries, and simulation specifications. [9]

The one aspect of Modelica that really stands out and differentiates Modelica from regular languages is that it enables acausal modeling, i.e. it is possible to describe the behavior of a component without defining which variables are input and which are output. As long as enough variables are specified Modelica solves the system of equations to assign the right values to the remaining variables.

Various formalisms can be expressed in the more general Modelica formalism. In this respect Modelica has a multi-domain modeling capability which gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. [9]
3.2 OpenModelica Environment

OpenModelica is a complete Modelica modeling, compilation and simulation environment based on free software. Notable components of the OpenModelica environment are OpenModelica Interactive Compiler (OMC) and OMShell. The OpenModelica Interactive Compiler is the core component of the environment and provides advanced interactive functionality for model management. The OMC functionality is available via command line scripting, CORBA or socket interface. [4] OMShell is an interactive command handler that provides very basic functionality for loading and simulation of models.

3.2.1 OpenModelica Modules

Figure 3.1 on page 12 contains an overview of the modules in the OpenModelica Compiler. We have chosen to emphasis on the modules we have modified during the course of this thesis.

- The Ceval module performs evaluation of expressions both at compile-time and interactively. It is among other things respon-
3.3 MetaModelica and RML

Relational Meta-Language (RML) is a meta-language and compiler generator, implementing Natural Semantics. [2] Natural Semantics is a specification formalism that is used to specify the semantics of programming languages. [8]

Initially the OpenModelica compiler was developed using RML. OpenModelica users usually have detailed knowledge of Modelica but little RML or Natural Semantics knowledge. To enable users to contribute to the open-source project OpenModelica the compiler was converted from RML to MetaModelica. MetaModelica is a modeling language based on Modelica with several extensions that allow program language specification. [9]
Part II

Incremental Compilation and Dynamic Loading of Functions
Chapter 4

Dynamic Loading of Functions

One of the problems given to us to solve was the low performance in evaluating function calls in OpenModelica. This was because external functions were compiled into executables, launched through the system shell and used text-files for variable passing.

All external functions in OpenModelica, except those called from simulation objects, were generated to C-code and compiled one by one into executables. External functions called from simulation objects were included in the simulation object’s C++-code and compiled together with the model into a simulation executable. When OMC called an external function the function was compiled and executed – each and every time.

The first step when improving the call performance was to compile external functions as dynamic libraries rather than executables, thus removing the penalties for launching new processes which on at least some platforms are expensive. See appendix B.1 on page 81 for benchmark data.

The second step was to make sure external functions were compiled only if needed during an instance of OMC using a cache to remember compiled and dynamically loaded functions. To handle function overloading, the cache is cleared when a loadFile call loads a Modelica file with a function using the same full name. In order to only compile external functions when a new implementation was given, we also needed to implement functionality to verify the currently compiled
4.1 Variable Passing

When calling functions from OMC, the variables had to be converted from MetaModelica/RML into C-based data types and then passed to and from the executable containing the function code. Simulation objects however handled their own variable passing directly in C-code as the simulation object generated all of its external functions code inside its own executable. Variable passing between OMC and a simulation also use text-files as the medium.

The third step when improving the call performance was to use direct-memory access for variable passing. Dynamically loading external functions into the OMC process made it possible to use direct memory access. Here we had to decide how we were going to encapsulate the data. Our options were to either use the already existing RML encapsulation or to build our own based on the old file-based encapsulation. The benefit of using the already existing RML encapsulation would be that we would not have to decapsulate and encapsulate the data as it is already passed through RML to C. The drawbacks were that we would have to make the RML headers a part of OpenModelica and thus increasing the dependencies. The benefits of building our own encapsulation would be that we could make a less complex solution. We would not need to change the code generation in OpenModelica as we could reuse the abstract data-types already defined. With this in mind we decided to build our own data encapsulation. More detailed information about the resulting conversion code and encapsulating of data can be found in chapter 8.1 on page 35.

4.2 Packages

The old implementation did not distinguish between different external functions – all were treated equally. One executable was created for each and every external function that was called within a project. This method creates a large number of files and many file operations are needed when compiling and loading the functions.
4.3 Overloading Functions

Introducing dynamic loading of functions did not remedy the problem of many files and file operations. As a solution we selected to group functions together in fewer and larger files using a language defined grouping: Top-Level Package. Package functions implemented in other languages, imported functions, are still handled as separate external functions. All other package functions are grouped together and need to be compiled as such – all at the same time.

4.3 Overloading Functions

OpenModelica has always had a problem with overloading of functions. If you have two top-level external functions that share the same full name in two different source files, they will overload each other. The old implementation solved this problem by always compiling the function when it was called. When introducing dynamic loading of functions we also tried to lower the number of compilations as they often were unnecessary. One problem we encountered while doing this was that non-package external functions creates a dynamic library based upon the full function name. When loading a source file any already loaded external functions with the same full names would be recompiled. In order to solve this we added a function that returns the source file for the external function, so when a source file containing external functions is loaded we can tell if any already existing dynamically linked library originates from the right source file or if it needs to be compiled.
Chapter 5

Incremental Compilation

To decrease the time spent compiling rather than developing during a large project, incremental compilation in various forms are often used. The OpenModelica environment uses C and C++ compilers to do the last step when assembling executables. These compilers have support for incremental compilation based on files. This is not sufficient as Modelica projects tend to have very large files containing many functions. Even if only one of these functions are modified the whole file needs to be recompiled. Our goal is to improve the incremental compilation in OpenModelica so that only modified functions are recompiled.

5.1 Virtual Functions

For incremental compilation at the function level we have to be able to replace references to functions in already generated and compiled code. This is where the concept of virtual functions come in handy. Our virtual functions are implemented using several associative arrays of function names and pointers.

The point of using these associative arrays is to make it possible to associate a function name with a newer version of that function without having to change any other code. The reason why no code needs changing is that all function-calls uses lookup to find out which function to call. Each external function will at the start of the function body lookup all of its dependencies. This has to be done at each function-call because between calls a loadFile may have changed
some dependency functions. These lookups are only run once inside a function as we can guarantee that \textit{loadFile} will not be called during the function-call. The associative arrays uses hashing techniques to remain reasonably fast but still dynamic and flexible. The reason why the associative array needs to be dynamic is because we want to be able to add new functions to a package incrementally. Flexibility is needed because we wish to be able to replace implementations of packages on the fly. OpenModelica does not require a file to be named after its package, thus different implementations of a package can exist in different files and might need to replace the other implementation when loaded by \textit{loadFile}.

\subsection*{5.1.1 Why Not Use Static Function Arrays?}

OMC is an interactive compiler. This means that OMC does not know which files that are going to be loaded during a session. Because of this it is complicated to use static package or function arrays for external functions in OMC.

Inside a package it is known at compilation time which functions that are present, so a static array is no problem. Using a fully static array would however mean that you have to do a full compilation when you add a new function. To remedy this you can use static indexes but a dynamic array. Using static indexes in a package would replace a hash-lookup with an array-lookup which is faster.

If one wishes to have a table of function pointers in each package you would still need either a hash-lookup for the package or a static index that gets initialized when the package is loaded, demanding that the package in question is loaded. This creates problems when you have cross-dependencies between packages.

Another solution is to let OMC have one large table for all external functions, both those in packages and those outside. This will remove the extra package lookup in the solution mentioned above. Each loaded package would get a “base index” in the large function table. We can however not guarantee that the package always gets the same “base index” as the same package can be used in different projects with different lists of packages. Also, this will mean we have to give a package a new “base index” if they fill their allotted slot, i.e. gets a new function. Then we also have to update all packages de-
5.1 Virtual Functions

Dependent on this package so that they use the new “base index”. None of this solves the problems with cross-dependencies between packages as packages when loaded still need to get the “base index” for all packages they are dependent upon.

Another problem arises if you want to be able to reuse libraries compiled using static hashes in-between OMC sessions. When we load a package we cannot be satisfied by the fact that the dll of the loaded package is more recent than its source file. We also need to check that any package dependency has not been recompiled. If the package dependencies are more recent we cannot be certain that the hash indexes are valid.

An example of this would be if we have two packages, pkg1 and pkg2. pkg1 is dependent on pkg2. In one OMC session both packages are compiled and the indexes are OK. We now remove the pkg2 dll-file and start OMC in a new session where only pkg2 is used which compile a new dll-file for pkg2. Next we load pkg1, the indexes in pkg1 are now invalid. This would be even more complicated if we have circular dependencies between pkg1 and pkg2.

So if one were to use static arrays you do not only need to change the dynamic hash to a static array, a lot of other code needs to be rewritten as well.

We did not have the time to pursue this fully so this is something that could use more investigation, see chapter 10.2.2 in page 61 for suggestions on which functions need to be modified.

5.1.2 Master Lookup

To facilitate the need for a lookup function we created a master_lookup. The master_lookup function has two associative arrays, one for packages and one for non-package external functions.

As we group all the external functions within a package together into one dynamic library we were able to put an associative array for that package along with a local lookup function in the same dynamic library. An overview is displayed in figure 5.1 on page 27.

The associative array for packages in master_lookup contains package names and a pointer to their lookup function while the associative array for external functions contains function names and their function pointers. Their interaction is displayed in figure 5.2 on page
5.2 Patches

Virtual functions makes it possible to implement incremental compilation at a function level since we can replace functions during runtime. Because we group all the functions in a package together, without patches the whole package would need to be recompiled even if only one function has been modified. To introduce incremental compilation at function level, we group modified functions together in patches and compile only those. When loading a package, all patches are automatically loaded in chronological order replacing function pointers in the associative array of the package. We load all patches chronologically because it is faster to replace a function several times than it is to find out which version is the latest and only load that one.

Let’s show this with an example. For an overview of the different components involved see figure 5.3 on page 28. In this example we assume that we have an editor that notices modifications and creates the necessary patch files to use incremental compilation. We have two packages, Pkg1 and Pkg2 each in its own Modelica source file, pkg1.mo and pkg2.mo respectively. Pkg1 has a function that calls Pkg2.function. Pkg2 has also a function that initially adds the two arguments.

We use the OMShell to input the following commands:

```plaintext
- - OMShell - - - - - - - - - - - - - - - - - - - - - - - - - - - -

loadFile("pkg1.mo")
loadFile("pkg2.mo")
Pkg1.function(1,2)
// ** Here we edit pkg2.mo **
loadFile("pkg2.mo")
Pkg1.function(1,2)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This results in the following steps for OMC:
1. OMC preloads both Pkg1 and Pkg2 after their respective `loadFile` calls.

2. OMC wants to evaluate `Pkg1.function` but realizes that neither Pkg1 nor Pkg2 are initialized or compiled.

3. OMC does a full compile of both Pkg1 and Pkg2 as Pkg1 has dependencies on Pkg2.

4. OMC initializes the resulting dll:s. When the dll:s are being loaded the pkghash in OMC gets two new entries. The key is a hash of the package name and the data is the address to the `lookup` function for that package.

5. OMC evaluates the function `Pkg1.function`.
   
   (a) OMC looks up the `lookup` function for Pkg1 using the pkghash and uses that function to find the function pointer to `Pkg1.function`.
   
   (b) OMC calls `Pkg1.function`.
   
   (c) `Pkg1.function` is dependent on `Pkg2.function` so `Pkg1.function` uses `master_lookup` to find out the function pointer for that function and stores that pointer for the remainder of the scope of the function. So even if `Pkg2.function` would be called inside a loop it would only need to lookup the function pointer once.
   
   (d) The function pointer is used to call `Pkg2.function` which is evaluated and return its result.
   
   (e) `Pkg1.function` can now be evaluated and returns its result to OMC.

   ** pkg2.mo is altered and `loadFile("pgk2.mo")` is executed. **

6. OMC removes the old Pkg2 entry from the pkghash as it has not yet made certain that the newly loaded Pkg2 is the same package or originated from the same source file as the old Pkg2.

   * `Pkg1.function` is called again *

7. OMC evaluates the function `Pkg1.function`. 
(a) OMC realizes that Pkg2 is not loaded yet.
   i. OMC notices that Pkg2.dll already exists but that pkg2.mo
      has been modified and that there exists a patch file
      indicating that we only need a partial compilation re-
      sulting in a patch dll instead of a full recompilation.
   ii. OMC loads the same old Pkg2.dll again.
   iii. The function hash in Pkg2 is instantiated with the
        function pointers for the functions in the old base dll.
   iv. The dll patch is loaded by the Pkg2.dll and the function
       pointer to Pkg2.function is updated in the Pkg2s own
       function hash.
   v. OMC adds Pkg2 to the pkghash again with the same
      old lookup function.

(b) Pkg1.function uses master_lookup to find out the function
    pointer to Pkg2.function.

(c) The function pointer is used to call Pkg2.function which is
    evaluated and returns its result.

(d) Pkg1.function can now be evaluated and returns its result
    to OMC.
5.2 Patches

Figure 5.1. Associative Arrays

Figure 5.2. Sequence for looking up functions
Figure 5.3. Package #2 is patched and pointers in both pkghash and Pkg1.function need to be updated
Chapter 6

Simulations

With dynamic loading of functions and incremental compilation of packages added to OpenModelica it was time to let simulations make use of it as well.

Simulations has always been handled differently than the other generated code in many aspects. First and foremost, they have always been compiled as stand-alone executables to make it possible to run them outside of OMC. We wanted to keep this particular feature as it can be very usable. As the simulations were compiled and executed outside of OMC they did not have to call external functions the same way, instead all of the external functions were collected into one single C++ file and compiled together. This removed the need for using text files for variable passing and thus simulations did not suffer from the speed penalties those brought.

Development of simulations that use a lot of external functions would benefit from the same incremental compilation that the rest of OpenModelica now utilizes. To be able to benefit from incremental compilation and at the same time keep simulations independent of OMC every simulation executable have to implement its own version of \texttt{master\_lookup} as the one in OpenModelica would not be available. The \texttt{master\_lookup} in simulations will however be simpler as no support for overloading is needed. This means we could separate all the actual simulation code from the external functions and their packages in such a way that if an external function was modified, only it need to be recompiled and if the simulation model is modified, only the main simulation executable need to be recompiled. We also made
sure that simulation executables are only recompiled if their model code has changed and not at every `simulate` call which was the case earlier.
Chapter 7

Extra Improvements

This chapter details other modifications we did during the thesis that are not directly connected to either dynamic loading or incremental compilation.

7.1 Arrays

To facilitate the communication between external functions and OpenModelica the module `read_write` is used. For simple data types such as real, integer or boolean nothing special is needed. Complex data types such as arrays are a different matter. All types of arrays need dynamic allocation and a lot of functionality such as transpose, sum and copy. String arrays and other arrays of complex types (arrays of arrays, tuples, or records are as of yet not supported) also need encapsulation of data. When we got involved in this project real arrays and integer arrays were in pretty good shape, boolean arrays seemed to be implemented and string arrays were non-existent. As we changed to using direct memory instead of using files as medium for transferring variables to and from external functions, support for boolean arrays, which had earlier been unsupported, came almost without charge. In the following days it became apparent that most of the boolean array functionality actually were stubs.

Later on a need for string arrays came into light and now it was time for an overhaul of the array functions. Quite a bit of the array code was, as you probably can imagine, very similar and instead of copying the code yet again for string and boolean arrays we modular-
ized it into a super array type; \texttt{base_array}. As this is all written in C, we could not use any language features to implement class hierarchies but the principle is still valid. So all common code got put into \texttt{base_array} and the interfaces for real and integer arrays got synchronized. Boolean and string arrays implements the same interface as real and integer arrays barring matrix and most vector operations. An identity matrix is not well defined for either boolean nor string arrays and neither is inner space products or even min/max/sum.

More in-depth information about \texttt{base_array} can be found in chapter 8.1.1 on page 36.

### 7.2 Records

OpenModelica has never had the functionality to transfer record variables to and from external functions. As we started making test cases and subsequently test which parts that were implemented and which were not we came to the conclusion that records had been skipped all the way from instantiation onto code generation. The type \texttt{record} was added into \texttt{Expression} and \texttt{DAE} modules so the code generator had something to work with. With the exception of finding where it was needed the addition of the type itself was simple enough. Only one additional member was needed in the record type compared to the primitive types and that was the class name of the record. This was very much needed in generation of the corresponding C struct for each and every type of record and what better to use as an unique identifier than the full class path of the record? It makes the code readable because it is easy to directly map the C struct to the Modelica record counterpart. Then it was just a matter of getting the code generator to generate code for the records as structs using the class definition of the record, change member access code and build a \texttt{read_write} container to handle sending arbitrary record data between the external functions and the \texttt{Values} module in OMC. Sending records to a function was not all that bad as both \texttt{System.executeFunction} (with a lot of help from \texttt{Values.Value}) and the external function code knows exactly what the record looks like and what members it has. When receiving the return value \texttt{System.executeFunction} is unaware of the record type and member variables. The solution for this was to make
the external function code send all needed type information back with the actual data.
Chapter 8

Implementation Details

In this chapter we will go into detail concerning our implementation.

8.1 Data Encapsulation

Data encapsulation is used when C and C++ code need to work on variables coming from or are about to be returned to OMC. The whole chain is actually: “Modelica function ↔ Values module ↔ RML ↔ System module ↔ read_write ↔ C or C++ code”. In this chapter we will only talk about “System ↔ read_write ↔ C or C++ code” as this is the part we have modified and worked with.

8.1.1 Modelica Types

Simple types such as integer, real and boolean are not encapsulated at all but instead directly mapped to the corresponding primitive types in C: int, double and signed char respectively.

Strings

Strings are mapped to char pointers as this is what C itself uses. These are however accompanied by utility functions to make it easier to allocate, deallocate and handle strings.
Arrays

While doing a cleanup of the array code in c_runtime we chose to split as much of the common array functionality as possible to work with a generic array class, base_array. More about why and a more high level explanation can be found in chapter 7.1 on page 31.

C has no language constructs for object-oriented programming but the concepts can still be used. So base_array is implemented as an interface but without use of virtual functions or the different access types: public, protected or private. For a full listing of the base_array interface see appendix A.1 on page 65.

All subclass arrays inheriting from the superclass base_array are really just typecasting. An instance of integer_array is just an base_array instance but every function in the integer_array interface knows that it should typecast the base_array data pointer member to a modelica_integer pointer before trying to access any elements. The same goes for allocation and deallocation of the data pointer member which is always handled by the subclasses. The base_array class implements the generic index, size and dimension function as those are common to all subclasses.

The subclass interfaces were kept intact barring inlining of a couple common functions and some minor cosmetic changes.

Tuples

Tuples are not encapsulated the same way other data types are. C or C++ functions returning multiple variables always does this using tuples but the C or C++ code has no concept of this. It just writes multiple variables to the read_write module and let it deal with it. See chapter 8.1.2 on page 37. Currently the code generators Codegen and SimCodegen does not support sending tuples to external functions.

Records

Records are mapped to C structs. No utility functions are needed as read_write handles all the work of sending this data between System and the C or C++ code. More about this in chapter 7.2 on page 32
and in the interface for **read_write** found in appendix A.2 on page 70.

### 8.1.2 The Bridge Between System and C/C++ Code

*read_write* is a collection of functions working with so called *type_description* objects. Together they form the bridge between the C code implementation of the *System* module and the generated C code of external functions. For the complete *read_write* interface, see A.2 on page 70.

When OMC calls an external function, *System.executeFunction* gets a list of input arguments encapsulated by RML and the *Values* module. From these arguments *System.executeFunction* builds a list of *type_description* objects by first decapsulating the arguments and then re-encapsulating them using the *write_modelica* functions from *read_write*.

The reason for this double encapsulation is primarily that *System.executeFunction* can not be written in such a way that it generically can call different function pointers with different arguments, thus we must encapsulate the arguments into a list that can be sent to a unique wrapper function for each external function. Secondly we do not reuse the *Values* and RML encapsulation as that would require the code generators to be rewritten in a non-trivial manner. All external functions would also need to link to RML and the *Values* module and by doing that increase the dependencies for external functions.

The unique wrapper function for each external function is generated by the code generators. These wrapper functions decapsulate the input arguments from *read_write* and send them to the real C function. The wrapper function then receive the return values from the real C function and encapsulates them before sending them back to *read_write* and *System.executeFunction*. As *System.executeFunction* receives the return values it decapsulates and re-encapsulates them again this time using RML and *Values*.

Multiple return values from an external function are always collected into a tuple, the generated C or C++ code has however no knowledge of this. The solution is to let an external function wrapper
call `write_modelica` a variable number of times. If `write_modelica` is called more than once the return value is put into a tuple.

This process does not in principle differ much from the old way of transferring arguments between external functions and OMC using text-files. The differences are that all transfer goes through direct-memory instead of files. Support for string arrays and records has also been added.
8.2 Function Cache

In the old version of OpenModelica, 1.4.3, there existed a list of external functions called cflist. When we started to implement dynamic loading of functions for the 1.4.4 version it was natural to continue using this list as a cache for compiled external functions.

The first new version only contained the full function path and a handle to the function pointer. So when OMC needed to call a function it had two possible scenarios:

- The function is in the cache so OMC only needs to call the cached handle.
- The function is not in the cache so OMC need to first compile the function, then call it and last save the resulting handle into the cache for use the next time the function is called.

With continued development and the introduction of packages and incremental compilation we needed to save more and more information in this function cache. The result became a list of CompiledCFunction entries.

8.2.1 CompiledCFunction: the Uniontype

Interactive.CompiledCFunction is a uniontype we have added containing two types of records: CFunction and VirtualFunction. These two record types represents external functions in different stages.

The CFunction record:

Absyn.Path path: The full function path so that the function can be found in the cache.

Integer funcHandle: A handle to the function pointer loaded in System. So OMC do not have to load the function again.

The CFunction represents non-package external functions that has been compiled.
The `VirtualFunction` record type:

- **Absyn.Path path**: The full function path so that the function can be found in the cache.
- **Option<Integer> funcHandle**: If the function is compiled the member contains a handle to the function pointer loaded in System. If the function is yet to be compiled the member is not set. A package function is added to the cache as soon as its package is loaded but is not compiled until one function in the package is called.
- **Option<Absyn.Path> pkg**: If the record points at a package function this member contains the full package path. Non-package external functions needs to be preloaded under certain conditions and for those this member is not set.
- **String pkg**: If the record points at a package function this member contains the directory path to the package source file. If it points at a non-package function, this is just the empty string.

The `VirtualFunction` represents several types of functions:

1. Preloaded package functions.
2. Compiled package functions.
3. Package functions that need recompilation.
4. Non-package functions that need recompilation.

Preloaded functions are needed because there is not enough information in order to instantiate the functions as their containing package are loaded. When calling one of these preloaded function the whole package is compiled and all records of functions in that package is upgraded to “Compiled package functions”.

Compiled package functions need to store more information than a non-package function after it has been compiled which is why we do not use the `CFunction` record type for these.

The reason why we need to be able to mark functions for recompilation is because of the possibility that functions and packages can be
replaced either with modified or completely new implementations. For example if we have two packages Pkg1 and Pkg2. Function Pkg1.f depends on Pkg2.f. Pkg1.f is called which forces OMC to compile both Pkg1 and Pkg2. Then Pkg2.f is altered and Pkg2 reloaded. When loading a package all items in the functions cache that matches that package path are removed and replaced by “Functions that need recompilation”. This is because if we would not mark the functions, that some other function is dependent upon, we could end up in a situation where Pkg1.f is called and as this function already is compiled we do not recheck all of its dependencies. As Pkg1.f is evaluated it would then not find Pkg2.f as it has yet been reloaded. To solve this we mark functions as “need recompilation” and after a loadFile or loadModel we look in the function cache for marked files and try to recompile them. We only need to do this after loadFile and loadModel as those are the only two functions that can load and replace packages and this means that we do not need to check the dependencies of already compiled functions every time the are called.

8.2.2 Functions

Almost every function we have written in the Meta-Modelica parts of OMC works with the function cache. It is the list around whom all other functions revolve – at least when it comes to dynamic loading of functions and incremental compilation.

These are the main functions that we have added to OMC that work with the function cache:

\texttt{Static.isFunctionInCfList}

Given a function path and the current function cache it will return an entry found matching the function.

It has three return values:

- Is function in cache? – true if it is, false if it is not.
- Function handle – optional, set if the function is in the cache and it is a compiled function.
- Package path – optional, set if the function is in the cache and is a package function.
Called by a lot of functions needing to know how to call a given function.

`Ceval.compileFunction`

Given the return values from a `Static.isFunctionInCflist` call and the current environment it tries to make sure the function is compiled.

This function was added as it became more and more complicated to handle all different types of functions available in the function cache. It handles these three different combinations:

- The function cache returned a handle to the function. Nothing needs to be done.
- The function cache returned a function path without a package. It is an external function that need recompilation, call `Ceval.compileExternalFunction`.
- The function cache returned a function path and a package path. It is either a preloaded package function or a package function that need recompilation. In either case, call `Ceval.compilePackageFunction`.

Called by a lot of functions after they have called `Static.isFunctionInCflist` in the search for a function handle.

`Ceval.compilePackageFunction`

Compiles the package library containing the given function and the current environment.

It calls `Ceval.compilePackageLib` to do the actual compilation and then it replaces every preloaded package function entry in the function cache with corresponding compiled function entry. After that it calls `Ceval.touchFunctions` to try and compile each and every dependency returned by `Ceval.compilePackageLib`.

Called by a lot of functions that needs a preloaded package function found in the cache to be compiled.
8.2 Function Cache

\texttt{Ceval \textit{compileExternalFunction}}

Compiles a function as a dynamic library given the full function path and the current environment.

It calls \texttt{Ceval \textit{compileExtFuncLib}} to do the actual compilation and then it adds an entry in the function cache with the handle to the compiled function. After that it calls \texttt{Ceval \textit{touchFunctions}} to try and compile each and every dependency returned by \texttt{Ceval \textit{compileExtFuncLib}}.

Called by a lot of functions that needs an external function not found in the cache to be compiled.

\texttt{Ceval \textit{touchFunctions}}

Tries to make sure a given list of functions is compiled using the function cache and the current environment.

For each function it does the following routine:

- The function is in the cache as either a compiled non-package function or a compiled package function. Nothing needs to be done.
- The function is in the cache as a preloaded package function, call \texttt{Ceval \textit{compilePackageFunction}} and continue.
- The function is not in the cache, call \texttt{Ceval \textit{compileExternalFunction}} and continue.

Called by \texttt{Ceval \textit{compilePackageFunction}} and \texttt{Ceval \textit{compileExternalFunction}} to try and compile all dependencies returned by \texttt{Ceval \textit{compilePackageLib}} or \texttt{Ceval \textit{compileExtFuncLib}}.

\texttt{Ceval \textit{compilePackageLib}}

Compiles a package library given the full path to the package, the source file from which the package definition originated and the current environment.

It does all the checking that is needed before determining if we need to either do a full compilation of the package, compiling a patch or doing nothing. Returns a list of all called functions from within functions inside the package, i.e. the package dependencies.
Called by `Ceval.compilePackageFunction`.

**Ceval.loadPackageLib**
Preloads a package given the full path of the package, its class definition, the path to the directory containing the package source files and the current environment.

Preloading a package means to fill the function cache with preloaded package function entries for each and every function inside the package.

Called by `ClassLoader.packageLoader` which in turn is called by `loadModel` and `loadFile`.

**Ceval.compileExtFuncLib**
Compiles a function as a dynamic library given the full function path and the current environment.

It does all the checking that is needed before determining if we need to either do a full compile of the library or do nothing. Returns a list of all function this function calls, i.e. its dependencies.

Called by `Ceval.compileExternalFunction`.

**Ceval.tryToRecompile**
Given the function cache and the current environment it will search for functions marked as need recompilation and compile those.

Called by `Ceval.cevalInteractiveFunctions` after it has evaluated either a `loadModel` or `loadFile` call.

**Interactive.removeAnySubFunctions**
Given a class that is about to be replaced, it will clear the function cache of all compiled entries and free any handles that are subitems to the class.

It is called by `Interactive.replaceClassInProgram` which in turn is called by `Interactive.updateProgram` when a `loadModel` or `loadFile` replaces a class in the environment.

**Interactive.removeCf**
Given an function that is about to be replaced, it will clear the
function cache of all compiled entries and free any handles that this function has.

It is called by `Interactive.replaceClassInProgram` which in turn is called by `Interactive.updateProgram` when a `loadFile` or `loadModel` replaces a class in the environment.
8.3 System

The dynamic loading of functions and packages needed a couple of low-level C functions to do the actual dirty work of loading and unloading the libraries. Also as generated C and C++ code need access to OMC's master_lookup this function and its associative arrays needed to be low-level as well.

8.3.1 Modelica Functions

This is a list of the new functions we added to the System module:

**System.loadPkg**

Loads and initializes a package library given the file path to the dynamic library (.so or .dll file) and the package path generated from its Absyn.Path.

If the dynamic library loaded OK it will add the package to the pkghash. Returns an handle to the library.

Called by Ceval.compilePackageFunction.

**System.loadExtFunc**

Loads and initializes a external function given the full path to the function generated from its Absyn.Path.

If the dynamic library loaded OK it will add the function to the exthash. Returns an handle to the library.

Called by Ceval.compileExternalFunction.

**System.lookupFunction**

Searches for a function inside a library given the library handle and a function name. Returns a handle to the function if found.

Every generated function has results in two functions. One real C or C++ function that does the actual calculations and only takes normal arguments and one external wrapper function that System.executeFunction uses when OMC want to call a function directly. This function is used by OMC to find this wrapper function.

Called by Ceval.compilePackageFunction and Ceval.compileExternalFunction.
8.3 System

System.freeFunction
Decrement the given function handle reference counter. A library is not freed before all its function handles have reached zero.

Called by Ceval.cevalCallFunction and Interactive.removeCf.

System.loadLibrary
Loads a library given its file path. Returns a handle to the library.

Not called anymore but does a small part of what System.loadPkg and System.loadExtFunc does.

System.freeLibrary
Decrement the given library handle reference counter.

Not called anymore but is a complement to System.loadLibrary.

System.removePackageFromHash
Removes a package from the pkghash given its package path. Will also free the associated library handles.

Called by Ceval.compilePackageLib and Interactive.removeAnySubFunctions.

System.removeExtFuncFromHash
Removes a non-package external function from the exthash given its function path. Will also free the associated function and library handles.

Called by Ceval.compileExternalFunction, Ceval.cevalCallFunction and Interactive.removeCf.

System.getPackageSourceFile
Returns the source file used to define the package when it was last compiled given a library handle.

Used to make sure the found package library originates from the same source file as the current package definition.

Called by Ceval.compilePackageLib.
System.getExtFuncSourceFile
Returns the source file used to define the function when it was last compiled given a library handle.

Used to make sure the found non-package external function library originates from the same source file as the current function definition.

Called by Ceval.compileExternalFunction.

System.getSimulationSourceFile
Returns the source file used to define the model when the simulation was last compiled given the full file path to the simulation executable.

Used to make sure the found simulation executable actually comes from the same source file as the current model definition does.

Called by Ceval.buildModel.

System.nameToPath
Converts a given string into an Absyn.Path.

Called by Ceval.compilePackageLib to parse function paths written to the patches file.

System.getLastPkgFile
Returns the last patch library given a file path to a package library.

Called by Ceval.compilePackageLib to figure out which is the last and thus newest patch library for a given package.

System.deletePatches
Given a file path to a package library it removes all patch libraries found.

Called by Ceval.compilePackageLib to make sure no old patches get reloaded when doing a full compile of a package.

System.getExeExt
Returns the extension for executables on the current platform. “.exe” on Windows and empty string on Unix.
8.3 System

Called by `Ceval.buildModel` to figure out what the compiled simulation executable is going to be called.

**System.getDllExt**

Returns the extension for dynamic libraries on the current platform. “.dll” on Windows and “.so” on Unix.

Called by a lot of function that need to know that package and non-package external function libraries are going to be called.

**System.setCompilerC**

Set the command used to compile C code into executables.

**System.getCompilerC**

Get the command to use for compiling C code into executables.

Called by `Ceval.generateMakefileHeader`.

**System.setCompilerCXX**

Set the command used to compile C++ code into executables.

**System.getCompilerCXX**

Get the command to use for compiling C++ code into executables.

Called by `Ceval.generateMakefileHeader`.

**System.setCFlags**

Set the flags to use with the command when compiling C code into executables.

**System.getCFlags**

Get the flags to use with the command when compiling C code into executables.

Called by `Ceval.generateMakefileHeader`.

**System.setLinkC**

Set the command used to compile C code into dynamic libraries.

**System.getLinkC**

Get the command to use for compiling C code into dynamic libraries.

 Called by `Ceval.generateMakefileHeader`. 
System.setLinkCXX
Set the command used to compile C++ code into dynamic libraries.

System.getLinkCXX
Get the command to use for compiling C++ code into dynamic libraries.

Called by `Ceval.generateMakefileHeader`.

System.setLinkCFlags
Set the flags used with the command when compiling C code into dynamic libraries.

System.getLinkCFlags
Get the flags to use with the command when compiling C code into dynamic libraries.

Called by `Ceval.generateMakefileHeader`.

System.setLinkSimFlags
Set the flags used with the command when compiling dynamic libraries to be used by an simulation executable.

System.getLinkSimFlags
Get the flags to use with the command when compiling dynamic libraries to be used by an simulation executable.

Called by `Ceval.generateMakefileHeader`.

System.setLinkSimRunFlags
Set the flags used with the command when compiling an simulation executable.

System.getLinkSimRunFlags
Get the flags to use with the command when compiling C code into dynamic libraries.

Called by `Ceval.generateMakefileHeader`. 
8.3 System

8.3.2 Internal functions

These are the new important functions we have added to the C code implementation of System module but that is not used or visible by Meta-Modelica parts of OMC.

master_lookup
For in-depth information about why and what master_lookup does see chapter 5.1.2 on page 23.

The two associative arrays pkghash and ext_vtable that master_lookup makes heavy use of are implemented by strhash_t added to c_runtime for which you can find the interface for in appendix A.4 on page 78.

load_library
Loads a library given a file path and returns a library handle.

Uses the platform independent functions for dynamic library handling collected under dynlib which interface can be found in appendix A.3 on page 77.

lookup_function
Lookup a function in a library given a library handle and a function name. Returns a function handle.

Also uses dynlib.
Part III

Result
Chapter 9

Benefits

In this chapter we will discuss the benefits of incremental compilation and dynamic loading of functions.

9.1 Dynamic Loading of Functions

A requirement to benefit from dynamic loading of functions is that the project uses external functions. The largest single benefit of utilizing dynamic loading of functions is the decreased runtime. The decreased runtime has its origin in decreased call time of external functions. A comparison between the methods used in OpenModelica can be found in Appendix B.2.1 on page 83.

Another factor is that OpenModelica used to compile external functions each and every time they were called. With dynamic loading this is only compiled if needed, thus the time saved increases with the number of times an external function is called.

The precision of real values has been improved because of the new variable-passing. When writing real values to file using a human readable format you lose precision so the improvement in precision is a result of passing variables through memory instead of files.

9.2 Incremental Compilation

Incremental compilation on a function level may decreases turn-around time when compiling during development of a Modelica project. First
time compilation of a package might take a longer time as all its external functions dependencies also need to be compiled. In earlier versions of OpenModelica only dependencies for called functions need to be compiled. In substantial projects, packet functions should be most common and developers work on a small subset of the whole project. Thus there is no need to recompile the whole project all the time and incremental compilation is a time saver.

Those who will benefit the most from incremental compilation are those who use an agile development method since the core in agile methods are small modifications and frequent testing.
Chapter 10

Conclusion and Future Work

In this chapter we draw conclusions based on the benchmarks of our implementation and give examples of future work.

10.1 Conclusion

This project has resulted in an OpenModelica version capable of dynamic loading of functions and incremental compilation on function level. Both of these features does reduce compile- and runtime. To be able to analyze the performance of complex features one needs good test cases and representative data. We have tried to find some larger real world Modelica projects but we were unable to locate any so we had to manufacture our own. See Appendix B.3 on page 85, B.4 on page 91 and B.5 on page 96.

10.1.1 Dynamic Loading of Functions

Dynamic loading of functions works very well and decreases runtime for external and package function calls from OMC. OMC used to use the System call to execute the executable generated when evaluating an external function, we now use dynamic loading which is \(~28\) times faster on Linux and \(~160\) times faster on Windows. Benchmark data can be found in appendix B.2.1 on page 83. Simulations already compiled all called functions as one single unit, already passing arguments through memory and did not need to call any external executables. As such dynamic loading of functions does not improve the compile-
or runtime of simulations. Dynamic loading of functions is however needed to use incremental compilation which simulations will be able to benefit from.

10.1.2 Incremental Compilation

With incremental compilation, virtual functions and dynamic loading all working together, the way OMC and Simulations call external functions has become very similar. This means simulations can benefit from incremental compilation just as OMC does.

In our benchmarks we compared runtime between three different versions of OMC:

1.4.3 has neither dynamic loading of functions nor incremental compilation.

1.4.4 has dynamic loading of functions but lack incremental compilation and collecting of external functions in packages.

Our version has both dynamic loading of function and incremental compilation of packages on the function level.

In most of our benchmarks our version turned out to be faster. All benchmarks first run an initial compilation that compiles all packages and functions from scratch. Then we run a number of consecutive compilations were we modify 3 to 4 functions in different packages each time.

On Linux we found out that if \( \sim 50\% \) of the functions in the project were used our version was \( \sim 1.3 \) times faster than OMC 1.4.3 and about as fast as OMC 1.4.4 during the initial compilation. In the consecutive compilations our version was \( \sim 2.5 \) times faster than OMC 1.4.3 and \( \sim 2 \) times faster than OMC 1.4.4. If \( \sim 90\% \) of the functions in the project were used our version was \( \sim 6.3 \) times faster than OMC 1.4.3 and \( \sim 5 \) times faster than OMC 1.4.4 during the initial compilation. In the consecutive compilations our version was \( \sim 9 \) times faster than OMC 1.4.3 and \( \sim 7.3 \) times faster than OMC 1.4.4.

On Windows we found the same general behavior although the differences in consecutive compilations were even larger. If \( \sim 50\% \) of the functions in the project were used our version was about as fast as OMC 1.4.3 and OMC 1.4.4 during the initial compilation. In the
consecutive compilations our version was $\sim 3.3$ times faster than OMC 1.4.3 and $\sim 2.9$ times faster than OMC 1.4.4. If $\sim 90\%$ of the functions in the project were used our version was $\sim 7.3$ times faster than OMC 1.4.3 and $\sim 5.8$ times faster than OMC 1.4.4 during the initial compilation. In the consecutive compilations our version was $\sim 12.6$ times faster than OMC 1.4.3 and $\sim 10$ times faster than OMC 1.4.4.

Generally the win in the initial compilation is because our version compiles packages as single unit compared with both OMC 1.4.3 and OMC 1.4.4 that compiles each function and its dependencies in their units. So as these benchmarks used many packages and functions calling each other both OMC 1.4.3 and OMC 1.4.4 compiled a lot of executables with duplicated function code inside them. The win in consecutive compilations is as expected from the incremental compilation on function level as our version does not need to recompile as many functions.

Simulations did not compile external functions as their own executable, so the difference for simulations are not as large although still very much notable. In our benchmarks we found the benefits for simulations to be about half of the benefits on most other projects, which still means a magnitude of $\sim 2$ or more.

We were able to find situations where incremental compilation always were slower. The most obvious one is if all functions are modified between each run but as the “Few functions used” benchmark on page 90 clearly shows it can also be slower if the number of unused external functions that the used packages depend upon is large enough.

### 10.2 Future Work

Our version of OpenModelica requires a *.patches* file to be able to tell which functions that need to be patched and thus use incremental compilation. Support for this need to exist in the development environment and can for example be achieved by a plug-in. It would however be nice if OpenModelica did not need this extra help from the development environment and could recognize which functions that had been altered all by itself. This is not an easy task and need thorough research.

It is plausible that a large number of package patches for a project
actually will make the run time slower due to the large number of file operations. It would be nice if OpenModelica could estimate when it would be more effective to remove all patches and do a full compilation.
10.2 Future Work

10.2.1 Static Array Function Lookups

As we discussed in chapter 5.1.1 on page 22 static hashes has a couple of problems but may be possible to use on a package level.

The dependency tracking is one of those things that need to be changed, and in our opinion this will require a lot of research in order to make sure that all functions calling a package has the correct and current index in the static array. This is not trivial when taking into account that OpenModelica can load packages dynamically and that compiled packages are supposed to survive between OMC sessions.

Our work would be a good base when implementing static array function lookups as all of the virtual function code and package encapsulation already is implemented.

10.2.2 Static Array Function Lookups: Implementation

The first things one need to change to be able to use static lookups are:

- `generateFunctionPtrInit` in `Codegen` that generates the C or C++ code for looking up a function. Instead of sending the package and function names to `master_lookup` send a static index taken from the function cache for example.

- `master_lookup` in `System` (systemimpl.c) that looks for the function pointer in OMC:s function tables. Instead of using the dynamic hashes, use a static table and lookup with index given above.

- `initPackageFunctions`, `preloadPackageFunctions`, `initExtFunc` all in `Ceval` that are responsible for adding entries to the function cache. If one chooses to use the function cache to store the static indexes, these are the functions that needs to be updated.

- `pkglib` in `c_runtime` that contains the package function hash and loads patches. Either replace the hash with a table or remove it and let OMC have all the function pointers. In the latter case, `pkglib_loadPatches` will need a way to update functions
in OMC. A solution similar to giving the package the function pointer to a `master_update` should be sufficient.

- `generateFullPkgLibHead` and `generateFullPkgLibFoot` both in `CodeGen` that generates the header and footer code for a package. You will probably need to update these functions if you do any changes to `pkglib`.

For simulations, that run their own process and thus has their own private `master_lookup` and hash these things need to be changed:

- `simulation_dynload` in `c_runtime` that contains a simulations very own implementation of `master_lookup` and function hash. Replace the function hash with a table in the same manner as in OMC.

- `generateSimulationCode` in `SimCodegen` that generates the main function for simulations. May or may not need changes.

The suggested changes above makes the assumption that when the code for a function is generated, `generateFunctionPtrInit` will be able to find the index for that function in the function cache. To make sure that is the case, some magic need to be installed in `compilePackageFunction` and `compileExternalFunction` in `Ceval` or any of their helper functions that loads the indexes for all dependencies before generating the function code. The problems are cross-dependent packages and packages not yet loaded. How to solve these we do not know.
Bibliography


[3] What are dynamic libraries?

[4] What is corba?

[5] Marsall Cline. What’s the difference between how virtual and non-virtual member functions are called?


Appendix A

Interfaces

A.1 base_array Interface

While creating the base_array interface we had it in our mind to mimic the already existing real_array, integer_array and boolean_array interfaces. Because of this the argument order varies a bit and some indexes are zero-based and some are one-based. The base_array has no destructor as all allocations are handled by a memory pool and all data in the pool allocated by a external function is freed as we leave the function.

```c
void base_array_create(base_array_t *dest, void *data,
                       int ndims, va_list ap)
```

Constructor. Initializes all members of a base_array.

- `dest`: a base_array object to initialize.
- `data`: pointer to the array data. Allocated by the subclasses as they know the size of each element. I.e. integer_array will allocate an array of integers but real_array will allocate an array of doubles.
- `ndims`: number of array dimensions.
- `ap`: variable argument list containing each dimensions size as an integer. As this constructor will always be encapsulated by the subclasses different constructors, we can not use ... here.
void simple_alloc_1d_base_array(base_array_t *dest, int n, void *data)
Constructor. Initializes a base_array as a 1 dimensional array (vector).

- dest : a base_array object to initialize.
- n : number of elements in the 1 dimensional array.
- data : pointer to array data allocated by subclass.

void simple_alloc_2d_base_array(base_array_t *dest, int r, int c, void *data)
Constructor. Initializes a base_array as a 2 dimensional array (matrix).

- dest : a base_array object to initialize.
- r : size of the first dimension or number of rows in the matrix.
- c : size of the second dimension or number of columns in the matrix.
- data : pointer to array data allocated by subclass.

size_t alloc_base_array(base_array_t *dest, int ndims, va_list ap)
Constructor. Initializes all members of a base_array except the data pointer. Returns the number of elements in the array given the number of dimensions and their sizes.

- dest : a base_array object to initialize.
- ndims : number of array dimensions.
- ap : variable argument list containing each dimensions size as an integer.

void clone_base_array_spec(base_array_t *source, base_array_t *dest)
Copy constructor. Initializes an array with the size and dimensions from another array. The elements are not copied.

- source : a source base_array object to copy from.
A.1 base_array Interface

- **dest**: a base_array object to initialize.

```c
void clone_reverse_base_array_spec(base_array_t *source, base_array_t *dest)
```
Copy constructor. Initializes an array with the dimensions and their sizes, but reversed, from another array. The elements are *not* copied. Given a 3 dimensional array with sizes 1, 2, 3 the resulting array will be a 3 dimensional array with sizes 3, 2, 1.

- **source**: a source base_array object to copy from.
- **dest**: a base_array object to initialize.

```c
size_t base_array_nr_of_elements(base_array_t *a)
```
Size accessor. Returns the number of elements in an array given its dimensions and their sizes.

- **a**: a base_array object.

```c
int ndims_base_array(base_array_t* a)
```
Size accessor. Returns the number of dimensions in an array.

- **a**: a base_array object.

```c
int size_of_dimension_base_array(base_array_t a, int i)
```
Size accessor. Returns the size of a specific dimension of an array.

- **a**: a base_array object.

```c
int base_array_ok(base_array_t *a)
```
Protected function. Runs sanity checks on an array. Returns non-zero if array is OK and zero if it is invalid in any way.

- **a**: a base_array object.

```c
void check_base_array_dim_sizes(base_array_t **elts, int n)
```
Protected function. Given an list of arrays it will assert if any of the arrays differ in dimension or size.

- **elts**: a list of arrays to check.
• \( n \) : number of arrays in the list.

```c
void check_base_array_dim_sizes_except(int k,
  base_array_t **elts, int n)
```
Protected function. Given an list of arrays it will assert if any of the arrays differ in dimension or size with exception for the dimension given by the argument \( k \).

• \( k \) : the dimensions for which the sizes are allowed to differ.
• \( elts \) : a list of arrays to check.
• \( n \) : number of arrays in the list.

```c
int base_array_shape_eq(base_array_t *a, base_array_t *b)
```
Protected function. Compares the dimensions and their sizes of two arrays. Returns non-zero if they are equal and zero if they differ.

• \( a \) : a base_array object to compare.
• \( b \) : another base_array object to compare.

```c
int base_array_one_element_ok(base_array_t *a)
```
Protected function. Returns true if all dimensions in the given array have size 1.

• \( a \) : a base_array object.

```c
size_t calc_base_index_spec(int ndims, int *idx_vec,
  base_array_t *arr, index_spec_t *spec)
```
Protected function. Returns an element index in the flat data array given an subscripted array and dimension indexes. It is used when handling index operations like \( A[3:7][2] \).

• \( ndims \) : number of dimensions in the subscript array and the index list.
• \( idx\_vec \) : list of dimension indexes. Each index is zero-based.
• \( arr \) : a base_array object.
• \( spec \) : a subscript specification.
A.1 base_array Interface

size_t calc_base_index(int ndims, int *idx_vec, base_array_t *arr)
Protected function. Returns an element index in the flat data array given dimension indexes. It is used when handling index operations like \(A[3,2]\).

- \emph{ndims} : number of dimensions in the index list.
- \emph{idx_vec} : list of dimension indexes. Each index is zero-based.
- \emph{arr} : a base_array object.

size_t calc_base_index_va(base_array_t *source, int ndims, va_list ap)
Protected function. Returns an element index in the flat data array given dimension indexes. It is used when handling index operations like \(A[3,2]\).

- \emph{source} : a base_array object.
- \emph{ndims} : number of dimensions in the index list.
- \emph{ap} : variable argument list containing indexes for each dimensions as an one-based integer. As this function will always be encapsulated by the subclasses, we can not use ... here.

int index_spec_fit_base_array(index_spec_t *s, base_array_t *a)
Protected function. Checks if the subscript specification fit the given array. Returns non-zero if it fits, zero if it does not.

- \emph{s} : subscript specification.
- \emph{a} : a base_array object.
A.2 read_write Interface

The new interface for read_write was inspired by the old interface. Most of the read_modelica and write_modelica functions are the same except that they now take a type_description object instead of a file-stream as their first argument. The old interface did not have any support for records and tuples was entirely handled by System, so the functions handling those types are new.

```c
void init_type_description(type_description *desc)
    Constructor. Initializes all members of a type_description object. An initialized type_description object is empty. Use the write_modelica functions to give it a value.
    • desc : a type_description object to initialize.

void free_type_description(type_description *desc)
    Destructor. Free all members of a type_description object.
    • desc : a type_description object to clear.

int read_modelica_real(type_description **descs, modelica_real *data)
    Given a list of type_description objects, take the first one and if it contains a modelica_real then return zero, set data to the value and increment the list one step. So if you want to read three modelica_reals, call read_modelica_real three times using the same list as argument. If the list is empty or the first type_description object in the list does not contain a modelica_real then a non-zero value is returned and the modelica_real pointer is left alone.
    • descs : a list of type_description objects.
    • data : a pointer to a modelica_real that will get the real value if everything went OK.

int read_real_array(type_description **descs, real_array_t *data)
    Given a list of type_description objects, take the first one and if it contains a real_array object then return zero, set data to point
A.2 read_write Interface

to the array object and increment the list one step. Otherwise an non-zero value is returned.

- descs : a list of type_description objects.
- data : a pointer to an real_array object that will be set to point at contained array object if everything went OK.

```c
void write_modelica_real(type_description *desc, modelica_real *data)
```
If the given type_description object is empty it will now contain the modelica_real value and return. If the type_description object already contains a value, the object will be converted to a tuple and the modelica_real value will be added as the last item in the tuple. If the type_description object already contains a tuple, the modelica_real will be appended at the end of the list of elements in the tuple.

- desc : a initialized type_description object.
- data : pointer to a modelica_real that should be added to the type_description object.

```c
void write_real_array(type_description *desc, real_array_t *data)
```
Set or add an array object to the given type_description object. If the type_description object already has a value, it is converted to a contain a tuple and the array object is added last.

- desc : a initialized type_description object.
- data : pointer to a real_array object that should be added to the type_description object.

```c
int read_modelica_integer(type_description **descs, modelica_integer *data)
```
Given a list of type_description objects, take the first one and if it contains a modelica_integer then return zero, set data to the value and increment the list one step. Otherwise an non-zero value is returned.

- descs : a list of type_description objects.
*data*: a pointer to a `modelica_integer` that will get the integer value if everything went OK.

```c
int read_integer_array(type_description **descs, integer_array_t *data)
```

Given a list of `type_description` objects, take the first one and if it contains a `integer_array` object then return zero, set `data` to point to the array object and increment the list one step. Otherwise an non-zero value is returned.

- *descs*: a list of `type_description` objects.
- *data*: a pointer to an `integer_array` object that will be set to point at contained array object if everything went OK.

```c
void write_modelica_integer(type_description *desc, modelica_integer *data)
```

Set or add a `modelica_integer` to the given `type_description` object. If the `type_description` object already has a value, it is converted to a contain a tuple and the `modelica_integer` is added last.

- *desc*: a initialized `type_description` object.
- *data*: pointer to a `modelica_integer` that should be added to the `type_description` object.

```c
void write_integer_array(type_description *desc, integer_array_t *data)
```

Set or add an `array` object to the given `type_description` object. If the `type_description` object already has a value, it is converted to a contain a tuple and the array object is added last.

- *desc*: a initialized `type_description` object.
- *data*: pointer to a `integer_array` object that should be added to the `type_description` object.

```c
int read_modelica_boolean(type_description **descs, modelica_boolean *data)
```

Given a list of `type_description` objects, take the first one and if it contains a `modelica_boolean` then return zero, set `data` to the
value and increment the list one step. Otherwise an non-zero
value is returned.

- **descs**: a list of type_description objects.
- **data**: a pointer to a modelica_boolean that will get the
  boolean value if everything went OK.

```c
int read_boolean_array(type_description **descs,
                       boolean_array_t *data)
```
Given a list of type_description objects, take the first one and
if it contains a boolean_array object then return zero, set data
to point to the array object and increment the list one step.
Otherwise an non-zero value is returned.

- **descs**: a list of type_description objects.
- **data**: a pointer to an boolean_array object that will be set
to point at contained array object if everything went OK.

```c
void write_modelica_boolean(type_description *desc,
                            modelica_boolean *data)
```
Set or add a modelica_boolean to the given type_description
object. If the type_description object already has a value, it
is converted to a contain a tuple and the modelica_boolean is
added last.

- **desc**: a initialized type_description object.
- **data**: pointer to a modelica_boolean that should be added
to the type_description object.

```c
void write_boolean_array(type_description *desc,
                         boolean_array_t *data)
```
Set or add an array object to the given type_description object.
If the type_description object already has a value, it is converted
to a contain a tuple and the array object is added last.

- **desc**: a initialized type_description object.
- **data**: pointer to a boolean_array object that should be
  added to the type_description object.
int read_modelica_string(type_description **descs, modelica_string_t *data)
Given a list of type_description objects, take the first one and if it contains a modelica_string_t then return zero, set data to the value and increment the list one step. Otherwise an non-zero value is returned.

- descs : a list of type_description objects.
- data : a pointer to a modelica_string_t that will get the string value if everything went OK.

int read_string_array(type_description **descs, string_array_t *data)
Given a list of type_description objects, take the first one and if it contains a string_array object then return zero, set data to point to the array object and increment the list one step. Otherwise an non-zero value is returned.

- descs : a list of type_description objects.
- data : a pointer to an string_array object that will be set to point at contained array object if everything went OK.

void write_modelica_string(type_description *desc, modelica_string_t *data)
Set or add a modelica_string_t to the given type_description object. If the type_description object already has a value, it is converted to a contain a tuple and the modelica_string_t is added last.

- desc : a initialized type_description object.
- data : pointer to a modelica_string_t that should be added to the type_description object.

void write_string_array(type_description *desc, string_array_t *data)
Set or add an array object to the given type_description object. If the type_description object already has a value, it is converted to a contain a tuple and the array object is added last.

- desc : a initialized type_description object.
A.2 read_write Interface

- **data**: pointer to a string_array object that should be added to the type_description object.

```c
int read_modelica_complex(type_description **descs, modelica_complex *data)
```

Given a list of type_description objects, take the first one and if it contains a modelica_complex object then return zero, set data to point to the object and increment the list one step. Otherwise an non-zero value is returned.

- **descs**: a list of type_description objects.
- **data**: a pointer to an modelica_complex object that will be set to point at contained object if everything went OK.

```c
void write_modelica_complex(type_description *desc, modelica_complex *data)
```

Set or add a modelica_complex object to the given type_description object. If the type_description object already has a value, it is converted to a contain a tuple and the modelica_complex object is added last.

- **desc**: a initialized type_description object.
- **data**: pointer to a modelica_complex object that should be added to the type_description object.

```c
int read_modelica_record(type_description **descs, ...)
```

Given a list of type_description objects, take the first one and if it contains a record then return zero, set each pointer given as variable arguments to the corresponding element in the record. If you have a record with three members, a modelica_integer, a modelica_string_t and a real_array you should call read_modelica_record with a modelica_integer pointer, a modelica_string_t pointer and a real_array_t pointer. One for each member.

- **desc**: a list of type_description objects.
- **...**: a variable length list of pointer arguments that point to each record member in turn.
void write_modelica_record(type_description *desc, 
const char *name, ...)
Set or add a record to the given type_description object. If the
type_description object already has a value, it is converted to a
contain a tuple and the record is added last.

You have a record in modelica whose full path is “Package.TestRecord”.
It contains three members, a modelica_integer called “count”,
a modelica_string_t called “description” and a real_array_t
called “terms”. To write this record the the type_description
object, you call write_modelica_record as this:
write_modelica_record(&desc, "Package_TestRecord",
TYPE_DESC_INT, "count", &(r.count), TYPE_DESC_STRING,
"description", &(r.description), TYPE_DESC_REAL_ARRAY,
"terms", &(r.terms), TYPE_DESC_NONE);
The last TYPE_DESC_NONE is to mark the end of elements.

- desc : a initialized type_description object.
- ... : a variable length list of member arguments ended by
  TYPE_DESC_NONE. See above.

type_description *add_modelica_record_member(type_description
*desc, const char *name, size_t nlen)
Adds a member to a record contained by a type_description
object with the given name. Returns a pointer to the newly
initialized and added members type_description object.

- desc : a type_description object containing a record.
- name : name of the member to add.
- nlen : length of the name to add in characters.

type_description *add_tuple_member(type_description *desc)
Adds a new element to a tuple contained by a type_description
object. Returns a pointer to the newly initialized and added
elements type_description object.

- desc : a type_description object containing a tuple.
A.3 dynlib Interface

As dynamic library handling differs between platforms, especially between Windows and Unix, we created a couple of platform independent functions that the rest of all the code can use and thus collecting all platform specific code in one place.

```c
os_lib_t os_load_library(const char *filename)
    Loads a dynamic library from the given filename and returns a handle. Returns NULL on error.

void os_free_library(os_lib_t lib)
    Free a dynamic library earlier loaded by `os_load_library`.

void *os_lookup_function(os_lib_t lib, const char *function)
    Lookup a function inside a library given a library handle and a function name. Returns a function pointer if successful or NULL if not.

const char *os_last_error()
    Returns a text string with the last reported error. The format of this message is platform specific but should always be useful. Will never return NULL.
```
A.4 strhash Interface

A dynamic string hash. The hash key must be a string. The data connected with each key can be of variable size for each strhash_t object. Used by System to implement pkghash and ext_vtable.

strhash_t strhash_new(size_t datasize, pagealloc_freefunc_t datafree, size_t prealloc)

Constructor. Returns a allocated and initialized hash object.

- datasize: the size in bytes of the data connected with each key in the hash.
- datafree: function pointer to a free function to be called when an element is removed from the hash. May be NULL which means the data will not be deallocated.
- prealloc: the number of items that the hash should preallocate. Ignored if less than 1.

void strhash_free(strhash_t strhash)

Destructor. Deallocates a hash object and all elements in it.

- strhash: a strhash_t object to destruct.

void strhash_clear(strhash_t strhash)

Removes and deallocates all elements in a hash object.

- strhash: a strhash_t object.

void strhash_set(strhash_t strhash, const char *key, const void *data)

Add or replace an element in a hash object. If an element with the same key already exists in the hash that elements data is deallocated and replaced with the new data.

- strhash: a strhash_t object.
- key: the key to associate the data with.
- data: a pointer to data to add to the hash.
A.4 strhash Interface

void *strhash_set_new(strhash_t strhash, const char *key)
Add or replace an element in a hash object. If an element with
the same key already exists in the hash that elements data is
deallocates and replaced with the new data. Returns a pointer
to the newly allocated element data.

- strhash : a strhash_t object.
- key : the key to associate the data with.

void *strhash_get(strhash_t strhash, const char *key)
Returns the data associated with the key in a hash object. If
no element with the given key exists then it returns NULL. The
returned pointer will be valid until the element is removed from
the hash.

- strhash : a strhash_t object.
- key : the key associated with the data.

const void *strhash_getconst(const_strhash_t strhash, const
char *key)
A constant version of strhash_get.

void strhash_remove(strhash_t strhash, const char *key)
Remove an element from a hash object and deallocates its asso-
ciated data. If a element with the given key does not exist in
the hash the function does nothing.

- strhash : a strhash_t object.
- key : the key associated with the data.

void strhash_take(strhash_t strhash, const char *key, void
*dst)
Remove an element from a hash object but instead of deallocat-
ing the data it is copied to the dst pointer. If a element with the
given key does not exist in the hash the function does nothing.

- strhash : a strhash_t object.
- key : the key associated with the data.
- dst : a pointer to receive the associated data. Must be at
least datasize large.
const char *strhash_get_key(const_strhash_t strhash, const char *key)
Returns the pointer to the key string of the element that has a matching key. Returns NULL if no such element exists in the hash.
  • strhash : a strhash_t object.
  • key : a key to search with.

int strhash_haskey(const_strhash_t strhash, const char *key)
Returns non-zero if an element with a matching key exists in the hash.
  • strhash : a strhash_t object.
  • key : a key to search with.

size_t strhash_size(const_strhash_t strhash)
Returns the number of elements in a hash.
  • strhash : a strhash_t object.

int strhash_empty(const_strhash_t strhash)
Returns non-zero if a hash has no elements in it.
  • strhash : a strhash_t object.
Appendix B

Performance Tests

B.1 Process Calling Overhead

B.1.1 Result

<table>
<thead>
<tr>
<th>Method</th>
<th>GNU/Linux Avg</th>
<th>GNU/Linux Min</th>
<th>GNU/Linux Max</th>
<th>Windows XP Avg</th>
<th>Windows XP Min</th>
<th>Windows XP Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Short function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System call</td>
<td>28.6</td>
<td>28.4</td>
<td>28.7</td>
<td>163</td>
<td>157</td>
<td>168</td>
</tr>
<tr>
<td>Execute call</td>
<td>12.4</td>
<td>12.1</td>
<td>12.8</td>
<td>21.2</td>
<td>20.2</td>
<td>21.7</td>
</tr>
<tr>
<td>Dynamic load</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Long function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>System call</td>
<td>2.68</td>
<td>2.27</td>
<td>2.90</td>
<td>25.4</td>
<td>25.3</td>
<td>25.5</td>
</tr>
<tr>
<td>Execute call</td>
<td>1.83</td>
<td>1.65</td>
<td>1.94</td>
<td>4.26</td>
<td>4.19</td>
<td>4.31</td>
</tr>
<tr>
<td>Dynamic load</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table B.1. Measurements of process calling overhead

Each value represents how many times longer it took to execute the same function using different call methods compared to the dynamic loading method. For example the system call method took an average of 28.6 times longer than the dynamic loading method on GNU/Linux and on Windows XP the same system call method took an average of 163 times longer than the dynamic loading method.
B.1.2 Procedure

The short function was a simple add operation using two integers as arguments and returning the sum as an integer. The long function sorted an array of 4096 integers, with no input arguments and no return value. The argument passing is a major factor for the results in the short function case but has no significance for the results of the long function case since no arguments were passed. We called the short function a 100 times more each test run because the execution time was significantly shorter for the short function than the long function.

The OpenModelica implementation used system call and files to send and receive variables. We were not interested in testing if files was slower than direct memory access, just the calls. So we used process arguments to pass input arguments and process exit code as return value. This would not have been a viable solution for OpenModelica since it needs more flexibility, but it is the fastest way to send data to any other process without using shared memory constructs that are very OS specific.

The system call method used the libc `system()` function on both GNU/Linux and Windows XP. The execute call method used libc `fork()` and `execv()` on GNU/Linux and `CreateProcess()` on Windows XP. Windows API has similar functions to `fork()` and `execv()` as well but we choose to use the most native function. The dynamic load method used `dlopen()` from libdl on GNU/Linux and `LoadLibrary()` on Windows XP. For each dynamic load call we opened and closed the library. In the real world one would cache those handles to speed up things even more but that seemed like an unfair advantage for this test.
B.2 External Function-call

B.2.1 Result

<table>
<thead>
<tr>
<th>Test</th>
<th>GNU/Linux</th>
<th>Windows XP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Min</td>
</tr>
<tr>
<td>Repeated calls, small arguments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenModelica 1.4.3</td>
<td>23.46</td>
<td>23.09</td>
</tr>
<tr>
<td>OpenModelica 1.4.4</td>
<td>1.059</td>
<td>1.010</td>
</tr>
<tr>
<td>Our version</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Few calls, large arguments</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenModelica 1.4.3</td>
<td>1.549</td>
<td>1.530</td>
</tr>
<tr>
<td>OpenModelica 1.4.4</td>
<td>1.079</td>
<td>1.077</td>
</tr>
<tr>
<td>Our version</td>
<td>1.000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Table B.2. Measurements of external function calls

Each value represents how many times longer it took to execute the same test case using different versions of OpenModelica. For example the repeated calls test took an average of 23.46 times longer using OpenModelica 1.4.3 compared to our version of OpenModelica on GNU/Linux.

B.2.2 Procedure

To test external function-calls extensively we created two test cases.

Repeated Calls, Small Arguments

In the first test case we wanted to compare call times for external functions between different versions of OpenModelica. To achieve this we created a test case that calls an external function 100 times sending two integers as arguments. This way argument passing will not be a major factor in the result. OpenModelica in its current state does not support functions without input arguments.
Few Calls, Large Arguments

In the second test case we wanted to compare efficiency of argument passing to and from external functions between different versions of OpenModelica. To achieve this we created three external functions dealing with arrays. One created the array, one reversed the array and the third one sorted the array. The generated array had 2048 elements of type real. We could not use larger arrays because of static memory limits in OpenModelica (all versions) and memory leaks in 1.4.3.
B.3 Incremental Compilation: Single Package

B.3.1 Result

Results for “Nested functions”, “Many functions” and “Few functions used” benchmarks can be found in figures B.1, B.2, B.3, B.4, B.5 and B.6 on pages 88, 89 and 90 respectively.

B.3.2 Procedure

Each case was created to measure performance of incremental compilation on function level compared to basic incremental compilation on file level. The effect of incremental compilation on a project depends on the structure of the project and also the development method used by the developers. Therefore we created three extreme project archetypes.

OpenModelica 1.4.3 and 1.4.4 only compile external functions that are called during a run, but need to compile those functions each run. OpenModelica 1.4.3 compiles called functions every time they are called. OpenModelica 1.4.4 compiles them once each run. Our version compiles all package functions and their dependencies the first run. In the consecutive runs modified functions are compiled into patches. External functions can not be patched but they only recompile when needed.

A run consists of starting OpenModelica with a script file that is evaluated.

Nested Functions

In this test we have 100 numbered package functions that each calls the next one, thus nesting them. The last function in the chain is modified prior to each run. The top function is called once during a run.

- OpenModelica 1.4.3 compiles all functions as a single unit each call.
• OpenModelica 1.4.4 compiles all functions as a single unit at each call because we modify and load the mo-file between each call.

• Our version compiles the whole package as a single unit on the first run and at each consecutive run a patch for the modified function is created prior to each call.

Many Functions
In this test we have 100 package functions that all call a basic arithmetic operator. Half of these functions are called during a run. One function is altered prior to each run.

• OpenModelica 1.4.3 compiles each function as a single unit every time it is called.

• OpenModelica 1.4.4 compiles each function as a single unit every time it is called because we modify and load the mo-file between each call.

• Our version compiles the whole package as a single unit on the first run and at each consecutive run a patch for the modified function is created at the first call of a package function after the last loadFile.

Few Functions
In this test case we have some package- and external functions. One of the package functions calls an external function. One package function is called during each run and that function does not call any external function. OpenModelica 1.4.3 and 1.4.4 will only compile the called function each run while our version compiles the whole package and all its external function dependencies.

• OpenModelica 1.4.3 compiles the called function and its dependency as a single unit each time it is called.

• OpenModelica 1.4.4 compiles the called function and its dependency as a single unit each time it is called because we modify and load the mo-file between each call.
B.3 Incremental Compilation: Single Package

- Our version compiles the whole package as a single unit and all its dependencies as their own entities on the first run. At each consecutive run we create a patch for the modified function and compiles all package dependencies because we modify and load the mo-file between each call.
Figure B.1. Benchmark of a single package with nested functions on Linux

Figure B.2. Benchmark of a single package with nested functions on Windows
Figure B.3. Benchmark of a single package using many functions on Linux

Figure B.4. Benchmark of a single package using many functions on Windows
Figure B.5. Benchmark of a single package using few functions on Linux

Figure B.6. Benchmark of a single package using few functions on Windows
B.4 Incremental Compilation: Multiple Packages

B.4.1 Result

Results for the three benchmarks can be found in figures B.7, B.8, B.9, B.10, B.11 and B.12 on pages 93, 94 and 95 respectively.

B.4.2 Procedure

The earlier benchmarks were rather small and limited to a single package so we wanted to show the benefits of incremental compilation in larger projects that might be better representatives of real world projects. To achieve this we created a collection of 30 packages, each in its own mo file. Each package contains 20 functions.

- In the first benchmark 50% of all functions are called. 15% of these are called from the script file. Each top call results in an average of 20 functions called with an average depth of 7 functions.

- In the second benchmark 90% of all functions are called. 60% of these are called from the script file. Each top call results in an average of 31 functions called with an average depth of 6 functions. The reason for the large percentage of functions called from the script file is that OpenModelica 1.4.3 and 1.4.4 does not allow the same function to appear twice in a call-tree thus we were limited when connecting functions.

- The third benchmark is a copy of the second benchmark with the exception that we have tried to put the whole benchmark inside one single script file and thus not restarting OpenModelica between each iteration. But the script file was too large for OpenModelica so we had to split the script file. We chose to split it into three larger parts and one initial part. When OpenModelica is restarted the function cache is cleared and all of the packages have to be loaded again. This becomes very visible on OpenModelica 1.4.4 in the graphs as each top coincide with a switch of script file. On our Windows machine we were unable
to get this benchmark to work on OpenModelica 1.4.3. This is probably because the Windows machine has less resources than the Linux machine used for this benchmark.
B.4 Incremental Compilation: Multiple Packages

Figure B.7. Benchmark of multiple packages using 50% of the functions on Linux

Figure B.8. Benchmark of multiple packages using 50% of the functions on Windows
Figure B.9. Benchmark of multiple packages using 90% of the functions on Linux

Figure B.10. Benchmark of multiple packages using 90% of the functions on Windows
B.4 Incremental Compilation: Multiple Packages

Figure B.11. Benchmark of multiple packages using 90% of the functions with fewer OpenModelica restarts on Linux

Figure B.12. Benchmark of multiple packages using 90% of the functions with fewer OpenModelica restarts on Windows
B.5 Incremental Compilation: Simulations

B.5.1 Result

Results for the two benchmarks can be found in figures B.13, B.14, B.15 and B.16 on pages 97 and 98 respectively.

B.5.2 Procedure

All other benchmarks tested evaluation of script files with external functions and packages. Another big part of Modelica is its simulations. The following benchmarks run a simulation with the same number of packages and functions as the earlier multiple package benchmarks.

- In the first benchmark 50% of all functions are called. 18% of these are called from the simulation model. Each top call results in an average of 15 functions called with an average depth of 5 functions.

- In the second benchmark 90% of all functions are called. 55% of these are called from the script file. Each top call results in an average of 33 functions called with an average depth of 6 functions.
B.5 Incremental Compilation: Simulations

Figure B.13. Benchmark of simulation using 50% of the functions on Linux

Figure B.14. Benchmark of simulation using 50% of the functions on Windows
Figure B.15. Benchmark of simulation using 90% of the functions on Linux

Figure B.16. Benchmark of simulation using 90% of the functions on Windows
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/

© Kim Jansson, Joel Klinghed