DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

JIT compilation to realize flexible data access in simulation software

Manuel Fasching

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

JIT compilation to realize flexible data access in simulation software

JIT Kompilierung zur Realisierung von flexiblen Datenzugriffen in Simulationsanwendungen

Author:	Manuel Fasching
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Sebastian Rettenberger, M.Sc.
Submission Date:	15th of March 2017

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15th of March 2017 Manuel Fasching

Acknowledgments

First, I want to thank my family for always believing in me and supporting me in various ways. A very special thanks goes to my girlfriend Isabella who had to take a backseat over the last weeks but never stopped motivating me.

Second, I want to thank my advisor Sebastian Rettenberger for the interesting discussions and all the constructive suggestions. I want to also thank Prof. Dr. Michael Bader for giving me the opportunity to write this thesis at the Chair of Scientific Computing.

Finally, a big thanks goes to all my friends friends, especially Andreas, Annika, Astrid, Christian, Ina and Stefan, for the steady encouragement and their understanding over the last months.

Abstract

Input data or scenarios of simulation applications are frequently defined by mathematical functions. These functions are often implemented in the application's code and are, hence, statically compiled. The dependency between scenario implementations and the actual application causes an adaption of the application's code for every new scenario. The primary goal of this thesis is the development of a methology, based on Just-in-time compiling, capable of dissolving this dependency and enhancing the flexibility of data initialization.

The absence of an easy but flexible solution was identified as the primary issue causing this dependency. A library capable of Just-in-time compiling scenarios was developed in order to solve it. This library was integrated in the simulation applications Shallow Water Equations and SeisSol. Tests validated the proper behavior and the aptitude for simulation applications of the developed library.

The results show that the flexibility of data initialization can be improved by JIT compiling without causing unacceptable drawbacks.

Contents

1. Introduction

1.1. Motivation

Scientific simulation applications use numerical methods in order to simulate physical occurrences. To start the calculations, they need an initial dataset as an starting point. This dataset, called scenario, contains, for example, initial values, fixed parameters or boundary data. Simple scenarios, heavily used in application development, are often described through mathematical functions. Most simulation applications are not limited to one single scenario. To achieve dealing with multiple scenarios, they are added to the application's code. The active one is conditionally chosen, for example, by a parameter. Figure 1.1 shows an example for two scenarios. Both provide a value for a given point. They only differ in their actual implementation.

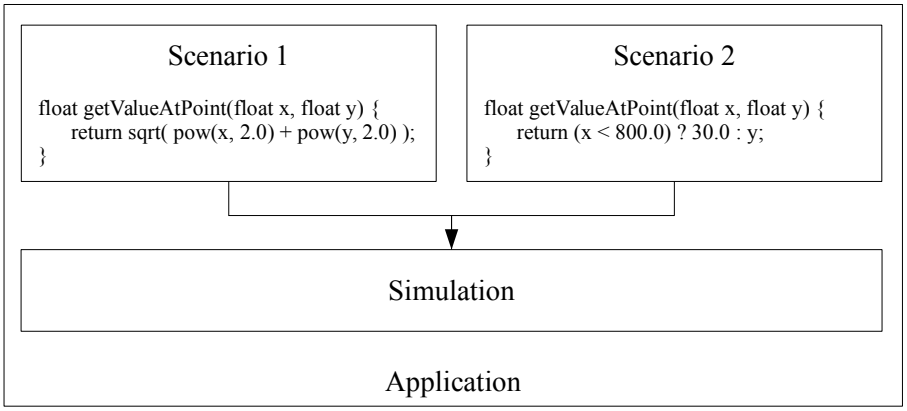


Figure 1.1.: Two scenarios of a simulation application.

The primary disadvantage of many scenario implementations is the low flexibility. The source code has to be adapted whenever a scenario is added or modified. This causes frequent compiling, especially in development and testing.
Another issue is the software design. Due to often historically grown code bases and high developer rotation, scenario maintanance may tend to be neglected. Usually, only few scenarios are implemented in an early development stage. The majority of scenarios is added later. The implementation of only a few scenarios does not necessarely require a sophisticated software design. Since developers want to focus on the simulation part,

it could actually be adviseable to keep it simple. The challenge is not to miss the point where this part of the application should be refactored. Listing 1.1 shows an excerpt of scenario implementations of the simulation application SeisSol. The "MaterialVal" vector is conditionally set based on another parameter which is implemented by a SELECT CASE statement. The whole source file has 2512 lines of code and consists of 25 CASE statements.

Listing 1.1: One case of the scenario definitions in SeisSol. Source code comments were deleted. [1]

```
1  [ ... ]
2  SELECT CASE(EQN%LinType)
3          CASE(0)
4                  MaterialVal(:,1) = EQN%rho0
5                  MaterialVal(:,2) = EQN%mu
6                  MaterialVal(:,3) = EQN%lambda
7                  IF(EQN%Advection.EQ.1) THEN
8                          MaterialVal(:,4) = EQN%u0
9                          MaterialVal(:,5) = EQN%v0
10                         MaterialVal(:,6) = EQN%w0
11                 ENDIF
12         [ ... ]
13 END SELECT
14 [ ... ]
```

This pattern heavily decreases the maintainability of this part of the application. Developers just focus on scenarios which they are in charge of and rather create new ones than reuse maybe already implemented scenarios. There is also no explicit scenario history or version management since a version control system only tracks on file level.

In summary, this thesis was motivated by two concrete issues in the data initialization phase of existing applications:

- The low flexibility of data initialization due to static scenario implemenations.
- An inappropriate software design for scenarios due to historically grown software and high developer rotation.

1.2. Issue Identification

Low flexibility and inappropriate software design were identified as two major issues of the data initialization part in simulation applications. In this section, the scenario implementations of two applications are going to be analyzed. Afterwards, each application is going to be mapped to a subset of the introduced issues.

1.2.1. Shallow Water Equations

The Shallow Water Equations (SWE) code is desribed as "An Education-Oriented Code for Parallel Tsunami Simulation". [2] SWE is written in C++ and implements currently six scenarios. Figure 1.2 illustrates the current scenario structure as a class diagram. The base

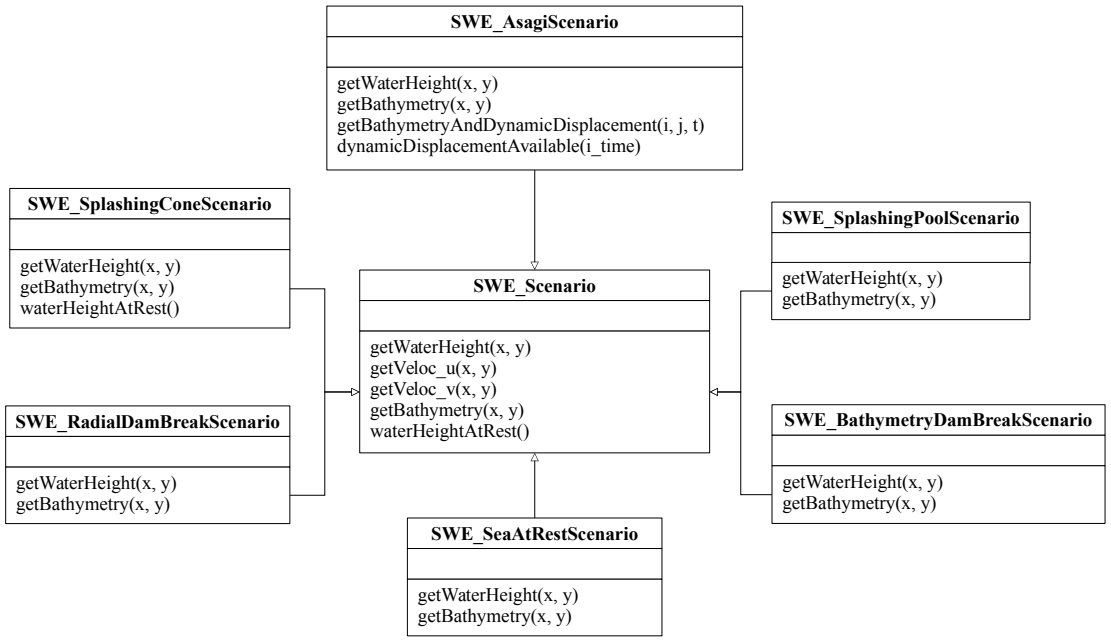


Figure 1.2.: Simplified UML class diagram of currently implemented SWE scenarios. [3]

class SWE_Scenario acts as an interface and also a very basic default scenario. The derived scenario classes override the functions according to their mathematical definitions. Listing 1.2 shows the getWaterHeight implementation of the SWE_RadialDamBreakScenario class as an example. The scenario part of SWE is designed according to the Strategy Pattern [4] which is considered as a convenient and clean software design. Since it acts as a teaching code, it is also well documented. The main disadvantage is the necessary recompilation if the user wants to switch the scenario. The build process is controlled by scons and the applied scenario is specified by a variable at compile time. This scenario choice mechanism heavily decreases the flexibility of the application.

1.2.2. SeisSol

SeisSol is desribed as "A software package for simulating wave propagation and dynamic rupture based on the arbitrary high-order accurate derivative discontinuous galerkin method (ADER-DG)". [5] Its data initialization consists of two parts, the initialization of the stresses on the fault and the material properties describing the seismic wave speed.

Listing 1.2: getWaterHeight function of the SWE_RadialDamBreakScenario class. [3]

```
1 float getWaterHeight(float x, float y) {
2         return ( sqrt( (x-500.f)*(x-500.f) +
3                 (y-500.f)*(y-500.f) ) < 100.f )
4                 ? 15.f: 10.0f;
5 };
```

The developers decided to implement these parts of the application in Fortran. The corresponding scenarios are defined in two files. The first one, containing the initialization routines for the stresses, consists of 4966 lines. The second one has 2512 lines. SeisSol is configured by a parameter file which is specified as an argument when SeisSol is invoked. Among others, it contains information for the selection of the right scenarios. Huge select case statements evaluate the information stated in the file and initialize the data. [1]

Such a design enables scenario selection after compile time which is currently not possible in SWE. However, it also has a lot of drawbacks in terms of software design. A lot of code changes are necessary in order to add new scenarios. This should be avoided since SeisSol is developed by several developers. Furthermore, a scenario is related to a specific configuration. The code stays untouched when a configuration is dropped, which potentially causes a lot of unused code.

1.2.3. Summary of Identified Issues

Two different methods how data initialization can be achieved were covered. Table 1.1 shows the analyzed simulation applications and the identified issues in order to give an overview. The primary reason for the current implementations is the absence of a simple solution which avoids all drawbacks. Application developers want to focus on the actual simulation. One approach to overcome the identified issue is Just-in-time (JIT) compilation.

Table 1.1.: Identified issues in the data initialization phase mapped to simulation applications.

Simulation application	**Identified Issues**
SWE	Flexibility
SeisSol	Flexibility, Software Design

1.3. JIT Compilation

In order to improve the flexibility, the scenario implementations have to be extracted from the application. To enable this separation, the application needs to provide an interface which can be fed with scenarios from an external source, like Figure 1.3 shows. The

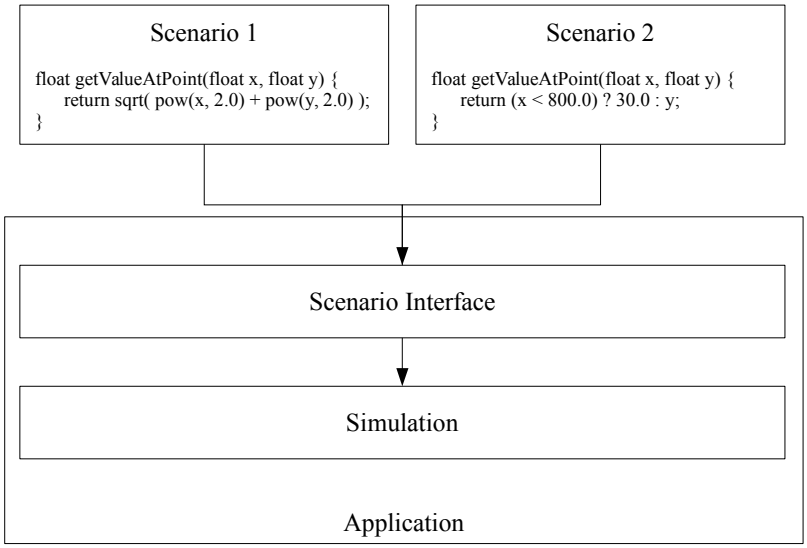


Figure 1.3.: Two application-independent example scenarios. The active scenario is loaded via an interface.

application just uses a well defined interface in order to intialize its data. The actual data is kept in external sources, for example in a library which is dynamically loaded. There is one drawback left when this approach is applied. If a scenario changes, compilation of it would be inevitable. To reach a maximum level of flexibility, the approach has to be extended by dynamically translating of scenarios during runtime. Therefore, a library is needed, which translates the scenario functions during runtime into machine code. Since a concrete mathematical function of a scenario is intended to be called multiple times, especially if dynamic adaptive methods are applied, interpreting would cause a huge performance loss. To avoid this bottleneck, the library has to enable JIT compilation. Figure 1.4 illustrates this idea. The source code of the chosen scenario gets translated into executable machine code which acts as source for the scenario interface. If the application wants to read a value, it just needs to execute the generated function. The execution can be repeated multiple times. The expensive step, the translation of the source code, must be done only once. All these steps are done during runtime which enables scenario changing without recompilation.

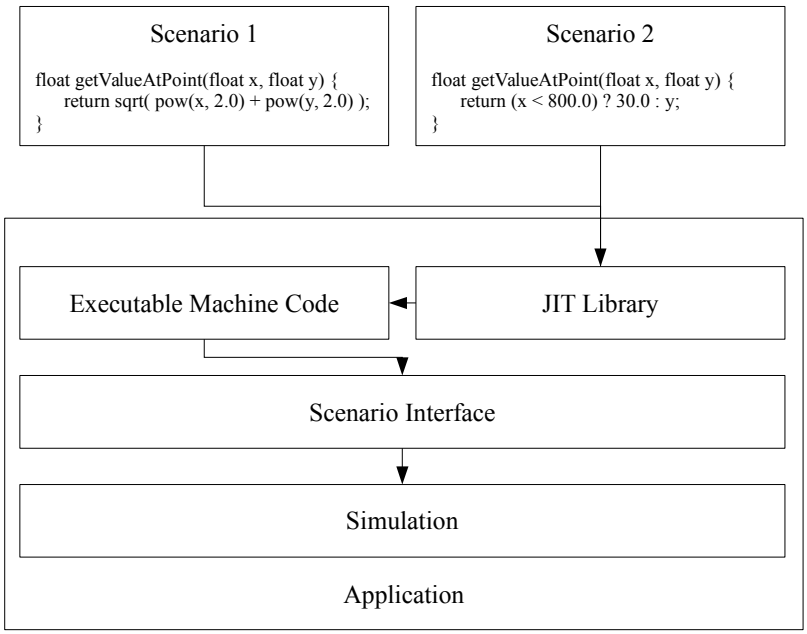


Figure 1.4.: Dynamically compiled example scenarios.

1.4. Summary

This thesis examines how flexibility in data initialization of simulation applications can be improved by JIT compilation without the introduced drawbacks. To fully understand the interests of the application developers, a requirements elicitation phase was conducted which is recorded in chapter 2.

Chapter 3 will convey the required theoretical concepts of compiler construction which are further used in this thesis.

Since there are many possibilities for compiler construction, chapter 4 will introduce related work and analyze their possible application in a JIT compilation library based on the elicitated requirements.

In chapter 5 the developed solution design and the applied concepts will be introduced.

Chapter 6 will show the integration of the library in simulation applications and the achieved results. Simulations were performed, in order to compare dynamic and static scenario loading and to validate the developed solution.

2. Requirements Elicitation and Determination

A library, like described in section 1.3, has the power to fully replace current scenario implementations and, hence, would have a big impact on software design of simulation applications. For this reason, the API is of major importance and it should encourage a software developer to use it. To get an understanding of important aspects and to elicitate the developer's requirements, expert interviews were conducted.

This thesis focusses on academic evaluation and demonstration of solutions for the issues identified in section 1.2. It is out of the scope of this work to develop a library capable of fully replacing the current data initialization methodology. However, it targets the most common operations, therefore, acts as starting point.

2.1. Expert Interviews

Requirements on the library obviosly differ based on the simulation application. Therefore, the interviewees were selected based on application ownership. Table 2.1 lists the selected persons and their owned applications.

The interviews followed a guideline which can be seen in appendix A. The guideline contains a list of questions, which were covered during the interview, and avoided loss of focus. The interviewees were highly encouraged to make hypotheses how such a library could be integrated in their simulation applications. Furthermore, the developer's requirements for the library were retrieved. Based on those two parts functional and non-functional requirements were inferred. The interviews were documented by audio capture.

Table 2.1.: Interviewees and their owned applications

Interviewee	**Simulation Application**
Sebastian Rettenberger, M.Sc.	SWE, SeiSsol
Carsten Uphoff, M.Sc.	SeiSsol
Dr. Vasco Varduhn	ExaHyPE

2.2. Inferred Requirements

2.2.1. Functional Requirements

Base Function

The library must be able to solve multiple mathematical equations which are combined in one function. The implementation of the function should be based on a well-known syntax, like C.

Basic Operations

The library should at least support the basic calculation types and the functions of the C Standard Math library.

Arguments and Local Variables

It must be possible to pass arguments to the function and to create and use local variables. An argument could either be a value or a pointer.

If/Else Statements

The library should be able to translate if/else statements. It should also be possible to nest them.

Arrays and Loops

It should be possible to deal with pointers in order to work on arrays. Therefore, loops are necessary.

Vectorization

Vectors as arguments should be accepted by the library. The function then applies to the total vector by vectorization and returns a result vector in order to increase the performance.

JIT Compilation

As described in section 1.3, the given function must be translated into executable machine code. The library returns only a function pointer to the memory address of the code.

Language Support

The targeted simulation applications are written in C++ and Fortran. The library should provide at least interfaces for those languages.

ASAGI

The library should be able to cooperate with ASAGI. [6] It should be possible to call ASAGI within the implemented functions.

2.2.2. Non-functional Requirements

Independence

The targeted simulation applications do not have dependency management. This means, all necessary dependencies must be installed by the user on her own. To keep the initial build process of the applications as simple as possible, the library should not depend on other 3rd party libraries. Moreover it should be possible to directly add the source code of the library to an application.

Usability

The library should provide a self-explanatory API with examples. Developers want to focus on their application, not on the library usage.

Extendability

The library should be extendable to various directions. Since it will generate executable machine code, it should be possible to supplementary add support of several instruction set architectures. On the other hand, the grammar should be extendable to special mathematical functions.

Lightweight

The library should focus on its main purpose and not unnecessarely inflate itself and, hence, the simulation application. Moreover, the compile time should be kept at a minimum level.

Performance:

The library will only be used for data initialization which usually constitutes only a small part of the whole simulation time. Thus, performance is not the most critical factor but it should not recognizably increase the simulation time. Since this is a very soft requirement and also heavily dependent on the simulation itself, it is assumed that the overall execution

time which is consumed for data initialization should as a maximum be 10 times higher than the execution time in the current state.

2.2.3. Exclusions

Semantic Analysis

The library does not have to perform a sophisticated semantical analysis. The developer is responsible to pass the right parameters and to implement the function correct, e.g., assign local variables before usage.

Types

As a starting point, the library will only support double precision floating point values. Other types are excluded.

Compiler Optimizations

Since performance requirements are not in the first place, compiler optimizations can be left out.

Target Architecture

The generated machine code will only be executable on machines supporting the x86-64 instruction set.

3. Theoretical Background

This chapter explains the theoretical aspects behind compiler construction. It is important to have a brief understanding of the different compiling phases and their purposes in order to get the idea of the JIT compilation approaches introduced in the next chapter. In detail, compiler construction basics, the relation to JIT compilation and virtualized environments are covered.

3.1. Compiler Construction

A compiler, irregardles if it is capable of JIT compiling, translates source code into machine code by passing the source code through multiple phases. Each phase transforms its input into a specified output format and hands it over to the next phase.[7] Figure 3.1 gives an overview of the phases and their outputs respectively their inputs. Error handling and table management, which relate to every phase, were omitted. The phases are structured in compiler frontend and compiler backend phases. Since a compiler produces machine dependent output, it is advantageous to seperate the machine independent part from the dependent one. Such a design fosters reusablity. When a compiler wants to support multiple architectures, it just needs to provide many compiler backends. The frontend stays the same. Vice versa, if a compiler wants to support multiple languages, it can provide multiple frontends and the backend stays the same. The following section explains the phases of the compiler frontend on the basis of the simple addition function shown in Listing 3.1. Since the Code optimization phase is optional and not relevant for the thesis, it is omitted.

Listing 3.1: Addition of two integers

```
1 int add(int x, int y){
2         return x+y;
3 }
```

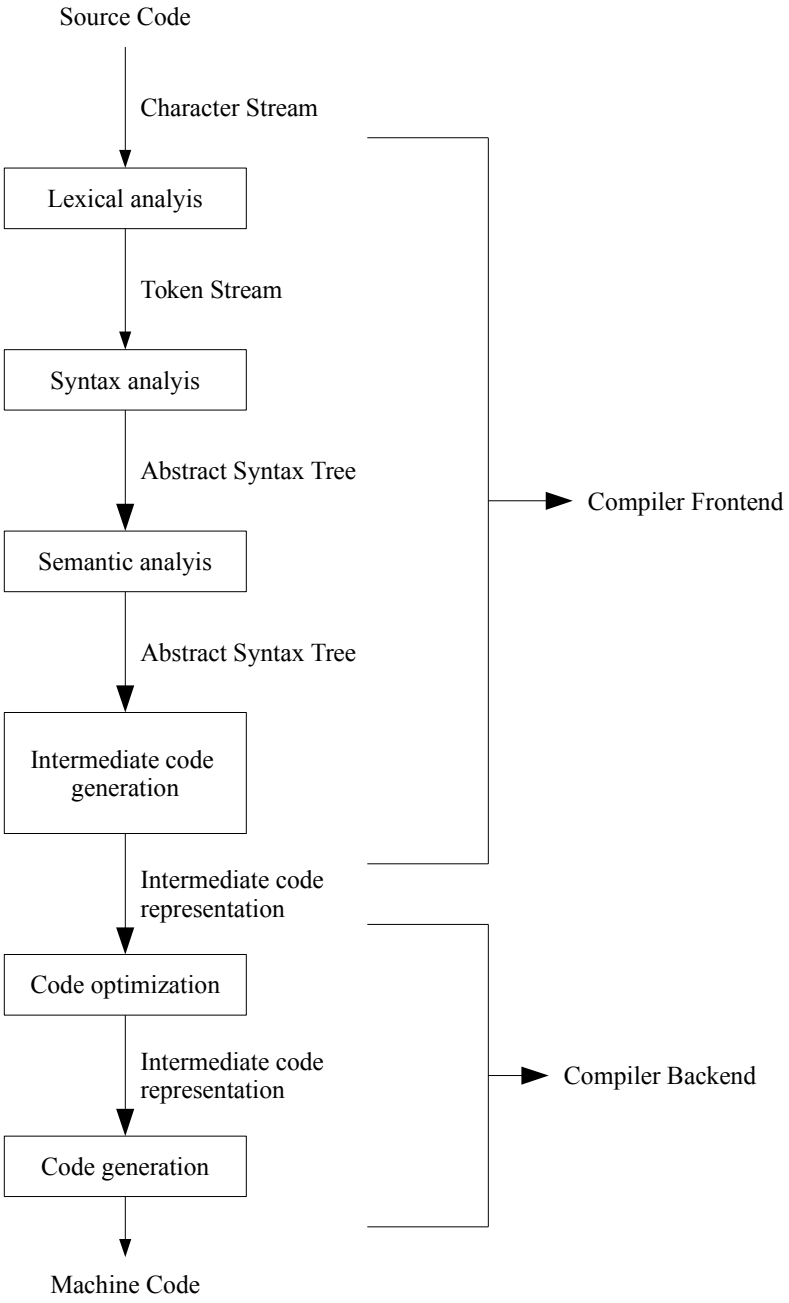


Figure 3.1.: Phases of a compiler. Error handling and Table management were omitted. [7]

3.1.1. Compiler Frontend Phases

The compiler frontend analyzes and structures the source code. Its output is a formalized code representation ready for the transfer into machine code by the backend.

Lexical Analysis

In the lexical analysis phase the input character stream is converted to a language specific token stream. A token is defined as a sequence of characters which form an entity. Tokens are clustered in categories. The specification of token categories is done by regular expressions. If a sequence of input characters matches, it is assigned to the corresponding category. Extraneous characters like comments or white spaces are dropped. In the end, all characters are part of a category, otherwise the source code contains unknown elements. [7] Table 3.1 shows exemplary categories and their corresponding matching rules.

If the rules are applied to the example addition function, a lexical analyzer would generate the follwing token stream:

$$[[INT\_TYPE, int], [IDENTIFIER, add], [SYMBOL, (], [INT\_TYPE, int], [IDENTIFIER, x], [SYMBOL, ,], [INT\_TYPE, int], [IDENTIFIER, y], [SYMBOL,)], [SYMBOL, \{], [KEYWORD, return], [IDENTIFIER, x], [PLUS\_OPERATOR, +], [IDENTIFIER, y], [SYMBOL, ;], [SYMBOL, \}]]$$

Table 3.1.: Example tokens and their rules

Category	**Matching Rule**
RETURN	return
INT_TYPE	int
IDENTIFIER	[A-Za-z][A-Za-z0-9_]*
OPERATOR_PLUS	+
SYMBOL	(,|;|{|}|(|))

Syntax Analysis

The syntax analyzer, also called parser, generates an abstract syntax tree based on defined grammar rules. It checks if the tokens conform to a structure defined by a context-free

grammar. There are also parsers dealing with context-sensitive grammers but they are not covered here.

A context-free grammar is defined as a 4 tuple. [7]

$$G = (N, T, P, S)$$

1. N means the finite set of nonterminal symbols
2. T means the finite set of terminal symbols
3. P are the productions of the grammar.
4. S means the start symbol.

Productions are specified as:

$$A \rightarrow \alpha, \quad A \epsilon N, \quad \alpha \epsilon (N \cup T)*$$

A context-free grammer in Backus–Naur form, which matches the example, could look like the following:

$$\begin{aligned}
N &= \{ < start >, < parameter_list >, < statement >, < expression > \} \\
T &= \{RETURN, INT_TYPE, IDENTIFIER, OPERATOR_PLUS, SYMBOL\} \\
P &= \{ < start > \ ::= \ INT_TYPE \ \ IDENTIFIER \ \ `(' < parameter\_list > `)' \\
&\qquad\qquad `\{' < statement > `\}' \\
&\quad < parameter_list > \ ::= \ < parameter > \mid < paramter_list > `,' < parameter > \\
&\quad < parameter > \ ::= \ INT_TYPE \ \ IDENTIFIER \\
&\quad < statement > \ ::= \ RETURN \ \ < expression > \\
&\quad < expression > \ ::= \ IDENTIFIER \\
&\qquad\qquad \mid < expression > \ \ OPERATOR_PLUS \ \ IDENTIFIER \ \ \} \\
S &= \{start\}
\end{aligned}$$

The example grammar is limited to only one statement per function, which has to be the return value.

Based on this grammar, the parser is able to generate an abstract syntax tree (AST). An AST represents the source code in a tree structure. It is important to mention, that it is not a concrete syntax tree which represents the grammar in a tree-like form. Information which is not needed for further analyizing and translation is dropped. The AST visualized in Figure 3.2 shows the transformed example.

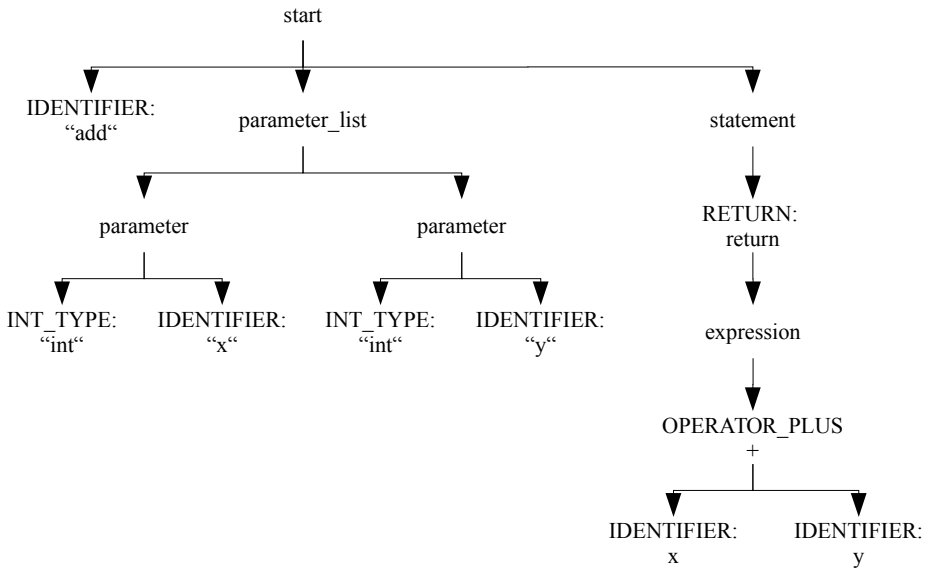


Figure 3.2.: The generated abstract syntax tree

Semantic Analysis

Even if the parsing has succeeded without errors, it is possible that the program still contains errors. A very common example is the access of a variable before it was initialized. The semantic analysis tries to find such errors. The types of performed checks highly depends on the specified language. [7] For example, in Java the semantic analysis must ensure that all exceptions are correctly catched, except of runtime exceptions. In most languages applied steps are:

1. Creation of the symbol table
2. Variable initialization before access
3. Type checking

Intermediate Code Generation

The AST produced by the parser is evaluated and transformed into a stream of instructions, called intermediate represenation (IR). There is no restriction regarding the format of the code. It is up to the developer to develop an appropriate intermediate code. Mostly it is based on assembly style. The program is also represented by instructions containing one operator and multiple operands. The main difference to assembly is the degree of abstraction. The IR does not need to specify dedicated registers. [7]

Listing 3.2: Addition of two numbers in x86-64 assembly

```
1 ; Move value stored in memory address 0x1 to rax register
2 mov  rax, [0x1]
3 ; Move value stored in memory address 0x2 to rbx register
4 mov  rbx, [0x2]
5 ; Perform addition of the values stored in rax and rbx
6 ; The result is stored in rax
7 add  rax, rbx
8 ; Move the value to memory address 0x3
9 mov  [0x3], rax
```

Listing 3.3: Addition of two numbers in machine code. The assembled assembly instructions are shown for explanation.

```
1 48 8b 04 25 01 00 00 mov rax, [0x1]
2 00
3 48 8b 1c 25 02 00 00 mov rbx, [0x2]
4 00
5 48 01 d8 add rax, rbx
6 48 89 04 25 03 00 00 mov [0x3],rax
7 00
```

3.1.2. Compiler Backend

In order to generate machine executable code, the IR is translated in machine instructions. The Assembly language is often used to describe those instructions. It consists of mnemonics which represent machine instructions, register names or flags. Its main target is the provision of a human readable form of machine instructions and to enable easier programming of them. [7] Listing 3.2 shows the assembly code for the addition of two integers stored at memory address 0x1 and 0x2. The result is written to memory address 0x3. Assembly heavily depends on the target architecture. The example was written for x86-64 machines, since it uses the 64 bit registers rax and rbx. The generated machine instructions would not be executable on a 32 bit machine.

In order to show an example for the generated output of the compiler backend, Listing 3.3 displays the assembled machine code in hex format together with the original instructions.

3.2. JIT Compilation

In comparison to Ahead-of-time compilers, JIT compilers emit machine executable code in memory instead of generating an executable file. This happens always during runtime which enables sophisticated optimization possibilities. Ahead-of-time compilers and JIT compilers basically have the same architecture, regarding the phases which a source code passes until it is fully translated in machine executable code. The major advantage a JIT compiler has, is the knowledge about the context. If the code gets translated during runtime, some accessed data is potentially already known when the compiling is invoked. This knowledge can be used to perform optimizations, e.g. to enhance constant folding, or to remove never reached conditional statements. On the contrary, an Ahead-of-time compiled code must be able to deal with all kind of allowed input data. It cannot make assumptions up front. [8]

3.3. Virtualized Environments

It is possible to virtualize whole runtime environments. Such environments usually take already precompiled code, called bytecode, as an input and translate it into machine code during runtime. This either happens through interpreting or JIT compilation. [8] Figure 3.3

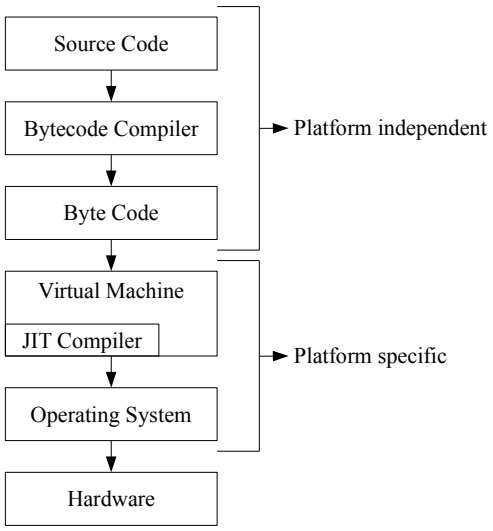


Figure 3.3.: The concept of virtualized runtime environments

shows the concept of virtualized runtime environments. The byte code compiler translates the source code into the byte code. Once the virtual machine is installed at a system, it is able to run the bytecode irregardless of subjacent operating systems and hardware. Since a compiler always generates machine code for a specified target architecture, this would

not be possible if Ahead-of-time compilation is applied. Another advantage of virtualized runtime environments is the enabling of high level features like garbage collection.

PyFR, a computational based fluid dynamics solver, demonstrates how virtualized environments can be used in high performance computing. It is written in Python, a language which is usually interpreted, but dynamically generates computation kernels. These kernels are defined in a C-like domain specific language and translated into machine code at runtime. [9]

4. Related Work

Two approaches were analyzed in order to increase the flexibility of data initialization. The first one is based on the idea of JIT compilation. Several possibilities, how a JIT compiler can be integrated in simulation applications are examined. The second one discretizes scenario data and dynamically loads them from an input file. This chapter explains both approaches and analyzes related work and their potential use in a JIT library.

4.1. JIT Compilation

The following sections analyze tools and libraries which are either applicable in compiler construction or provide an interface for dynamic compilation of C Code.

4.1.1. Compiler Construction

Flex

Flex is a command line tool and capable of generating lexical analyzers in either C or C++ code. Flex itself is not applied in compilers but its generated analyzers can be used for lexical analysis. It claims a well defined specification as an input. Basically this specification defines the tokens and their matching rules. Listing 4.1 shows a simple specification file of a line and character count program. A specification consists of the following three sections, which are seperated by a "%%" symbol: [10]

1. The definition section.
   Definitions which are later used in the rules section are placed here. It is also possible to add code which should appear at the beginning of the generated lexical analyzer, e.g. include statements.

2. The rules section.
   It contains a series of regular expressions and the desired actions. Formely defined definitions can be used in order to simplify the rules.

3. The user code section.
   Code placed in the user code section is copied to the generated lexical analyzer.

The example defines count variables for lines and characters in the definitions section. The specified rules define that the counts get incremented when a matching symbol is found.

Listing 4.1: Flex specification file of a line and character count program [10]

```
1  int num_lines = 0, num_chars = 0;
2
3  %%
4  \n ++num_lines; ++num_chars;
5  . ++num_chars;
6
7  %%
8  main() {
9      yylex();
10     printf( "# of lines = %d, # of chars = %d\n",
11                 num_lines, num_chars );
12 }
```

The generated lexical analyzer would be executable, since the user code section contains a main method.

Since flex allows the integration of user defined C/C++ code in its specification files, it is easy integrable in other software, e.g. a compiler.

GNU Bison

GNU Bison is a command line tool capable of generating parsers in either C or C++ code based on a specified bison grammar file. It is important to mention that Bison only deals with context-free grammars. A Bison grammar file consists of the following four sections:

1. C declaration
   In order to concatenate the parser with further components, C code like variable declarations or include statements can be placed here. The code will later appear at the beginning of the generated parser.

2. Bison declarations
   The symbols used in grammar rules are defined here. Compared with context-free grammars in Backus-Naur form, these are terminal and nonterminal symbols.

3. Grammar rules
   This section carries bison grammar rules represented in a machine readable backus-naur form.

4. The additional C code section
   Just like in flex specifications, Bison also offers the possiblity to integrate C/C++ code in its grammar files.

In order to concatenate Flex and Bison it is meaningful to define the tokens, generated by the lexical analyzer as terminal symbols, in the Bison declarations section. As stated in 3.1.1, a parser should map the source code to a tree structure. This can be achieved by declaring the tree elements in the C declarations section. The actual tree building happens in the grammar rules sections. They also carry actions for applicable rules.

Bison parsers also have a basic error handling, which outputs an error message containing the not parseable symbol and its position in the source code. [11]

LLVM

The LLVM project consists of many compiler and toolchain technologies. This passage only relates to the LLVM Core project.
LLVM's primary target is the provision of a reusable compiler backend for each kind of frontends. The interface is represented by an LLVM specific intermediate language - LLVM-IR. LLVM further optimizes and translates the code in machine executable code for the specified target architecture. The compiler does not need to emit code in target specific machine language, this part is done by LLVM. [12]

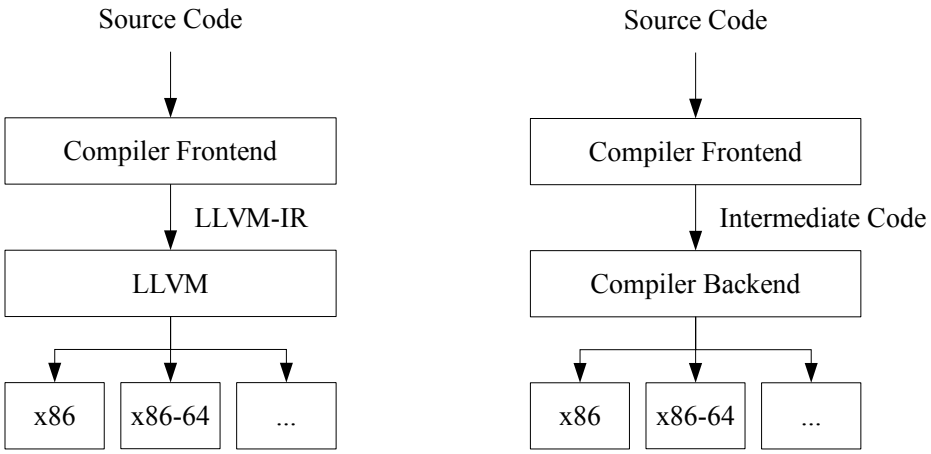
Figure 4.1 highlights the difference between a compiler architecture built on LLVM and respectively not built on LLVM. The major difference is the output of the compiler frontend. When LLVM is used as backend, it has to generate LLVM-IR code.

It also offers the implementation of JIT compilation specific functionalities, e.g. emitting machine executable code directly in memory, which makes it attractive for simulation applications. Using LLVM in a compiler backend has two big advantages. First, a compiler just needs to translate its input to LLVM-IR language. The further code generation for several architectures is done by LLVM. Second, it performs sophisticated optimizations on the LLVM-IR code, which is also a critical part for most compilers.

DynASM

DynASM is considered as a dynamic assembler for code generation engines. It is not a library in the general sense, but it provides a toolchain for creating your own code generator. DynASM is maintained as a part of the LuaJIT project. [13]

In comparison to LLVM, DynASM has not defined its own intermediate language but it abstracts assembly instructions. The developer writes instructions in an "assembly like" syntax which are further called DynASM instructions. Then, she invokes a preprocessor which generates C code out of those DynASM instructions. This code and some DynASM header files constitute a full code generation engine. DynASM is not capable of optimizing code or providing a platform independent interface like LLVM does. The developer has to take care of optimizations and platform constraints by himself. Listing 4.2 shows a simple example of DynASM instructions. It is especially possible to mix C/C++ code and DynASM instructions. The lines, which are getting replaced by the preprocessor, are



(a) Compiler architecture, using LLVM

(b) Compiler architecture, without LLVM

Figure 4.1.: Comparison of compiler architectures

Listing 4.2: Simple example of a to be preprocessed code snippet [13]

```
1 if (ptr != NULL) {
2   | mov eax, foo+17
3   | mov edx, [eax+esi*2+0x20]
4   | add ebx, [ecx+bar(ptr, 9)]
5 }
```

marked by a pipe symbol. The preprocessor takes those lines and translates them into C function calls.

Listing 4.3: Preprocessed DynASM code [13]

```
1 if (ptr != NULL) {
2   dasm_put(Dst, 123, foo+17, bar(ptr, 9));
3 }
```

As Listing 4.3 shows, the generated function call gets, among others, "123" as an argument. This number represents the offset in an action list which contains the partially specified machine executable code. The last two arguments are used to fill the gaps. Those were also present in the former DynASM instructions. In order to generate the fully specified machine executable code, the targeted instructions in the action list are combined with the arguments passed by the dasm_put function. This design enables the usage of C

expressions in DynASM instructions.

4.1.2. JIT Compilation Libraries

Apart from constructing a compiler, it would also be possible to use an already existing JIT compiler. They are mainly used in virtualized environments, as mentioned in 3.3. Due to the characteristics and possibilities of C, e.g. direct memory access, a virtualized environment targeting C applications makes no sense. This is also the reason why there are only few libraries which support JIT compiling of C Code.

However, one library was found and is introduced in the following.

Tiny C Compiler

The Tiny C Compiler (tcc) is a very light-weight, full ISO C99 compliant C compiler, developed by Fabrice Bellard in 2004. It targets machines with only little memory. Sine the executable for x86 code, containing C preprocessor, C compiler, assembler and linker, only has 100KB, it is even possible to run it on rescue discs. Tcc itself does not support JIT compilation but libtcc, the C library of tcc, can be used for dynamic code generation. [14]

Listing 4.4: Dynamic compilation of a C function with tcc [14]

```
1  char add[] =
2  "unsigned␣add(unsigned␣a,␣ungsigned␣b)\n"
3  "{\n"
4  "␣␣␣␣return␣a+b;\n"
5  "}";
6
7  int main(int argc, char **argv)
8  {
9      TCCState *s;
10     unsigned (*func)(unsigned, unsigned);
11     s = tcc_new();
12     tcc_set_output_type(s, TCC_OUTPUT_MEMORY);
13     /* Compile the function */
14     if(tcc_compile_string(s, add) == -1)
15         return 1;
16     /* Relocate the code */
17     if (tcc_relocate(s, TCC_RELOCATE_AUTO) < 0)
18         return 1;
19     /* Extract the function */
20     func = tcc_get_symbol(s, "add");
21     if (!func)
```

```
22          return 1;
23      /* Prepare variables and invoke the code */
24      unsigned a = 10;
25      unsigned b = 5;
26      printf("%u + %u = %u", a, b, func(a, b));
27      /* Delete the state */
28      tcc_delete(s);
29      return 0;
30  }
```

Listing 4.4 shows a minimum working example of tcc's dynamic compilation capabilites. In line 14, a char array, containing the function definition of an addition function, is passed to tcc. Line 20 shows how the function pointer to the executable code can be retrieved. In line 26 the dynamically compiled function is invoked. The example would produce "10 + 5 = 15" as an output.

Basically, tcc would fit as a dynamic compilation library in simulation applications. In order to provide a customized interface, it would be necessary to either use tcc as a base and build an interface layer on top or to develop a library which depends on tcc. Since the latter one would break the independence requirement, only the first approach would be an accepted solution. Apart from that, Fabrice Bellard stopped working on tcc in 2013.

4.2. Discretization

A completely different approach, in comparison to JIT compiling, is the provision of data based on values read from an input file. This approach excludes the possibility to describe scenarios directly by a mathematical function. It would be necessary to prior discretize the function and provide those values in an input file. Using a native approach, this implicates one primary challenge. When using dynamic adaptive grids, each node would have to load the entire file in its memory since the initially assigned area of a node may change. When dealing with big files, this would be impossible, respectively waste a lot of memory. A possible solution for that issue is the distribution of values to all nodes. If the application requests a locally not available value, it would have to dynamically load it from another node.

A library, which exactly targets that issue, is ASAGI. It provides an simple interface for the retrieval of distributed scenario data. This also improves the flexibility of data initialization since the scenario data is loaded by an external input file. [6]

There are many cases where scenario data is only available in a discretized form. For example, when the data were collected by measurement. If this is the case, the approach is definitely advantageous. For scenarios which can be simply described through mathematical functions, the prior discretization would produce additional work. Especially

in application development this is not desired.

4.3. Summary

Several tools whose usage are beneficial in compiler construction were covered. By means of tcc, there was also an entire compiler library which could be used. The discretization approach showed that JIT compilation is not the only way for solving the issue. In the end, for enabling JIT compilation, the decision between tcc and the construction of an own compiler must be made.

Using tcc brings a lot of drawbacks. As stated, tcc must be adapted in order to meet all requirements. Even though it has a very small size it does contain a lot of unneccessary features. This would decrease the extendability since that code always has to be maintained when the library gets extended, for example if ported to a new architecture. The original author of tcc has also stopped working on the project which means that there will be no updates from his side.

Considering these drawbacks, starting from scratch and constructing an own compiler was chosen as an appropriate approach. The parser and lexical analyzer generators are certain since GNU Bison and Flex are well-proven standard tools and often used toghether. Both tools come with a detailled documentation and many examples.

Regarding the compiler backend, both LLVM and DynASM have advantages and drawbacks. LLVM is capable of emitting machine code for many architectures and also performing optimizations on the code. Unfortunately, it would mean a dependency of the library which is excluded by the requirements. In contrary, DynASM generates an independent code generator which only carries translated self written code. Since the implemented library should be as lightweight as possible, this is a big advantage. In the first step, only the x86-64 instruction set must be supported. That means, automatic code generation for several architectures is not a requirement. Performance is also not a priority in the first place which invalidates LLVM's optimization capabilities. Combining all pros and cons, DynASM was chosen as tool for code generation.

Table 4.1 summarizes the advantages and drawbacks of the analyzed tools. Tools, chosen for compiler construction in this thesis, are highlighted bold.

Table 4.1.: Drawbacks and benefits of analyzed tools. Tools, chosen for compiler construction are highlighted bold.

Tool	**Purpose**	**Benefit**	**Drawback**
Flex	Lexical Analyzer	Well-proven, Integration capabilites	-
GNU Bison	Parser	Well-proven, Integration capabilites	-
DynASM	Code generator	Lightweight, Simple	Low abstraction
LLVM	Code generator	Optimizations, Architecture support	Library dependency
tcc	Compiler	-	Decreases extendability

5. Design

The developed library is intended to JIT compile numerical functions wich define a scenario of a simulation application and was called ImpalaJIT. Section 5.1 describes the high-level design of the library. The configuration possibilities of ImpalaJIT are described in section 5.2. Section 5.3 explains the interface and its usage. A detailled description of the different compiler phases is stated in sections 5.4, 5.5, 5.6 and 5.7.

5.1. Basic Concept

ImpalaJIT is written in C++. Its software design is based on the different phases in compiler construction, introduced in chapter 3. Since it needs not to support several instruction set architectures but only x86-64, the intermediate code generation was skipped. Figure 5.1 shows the aligned components of ImpalaJIT. The driver manages the entire flow beginning

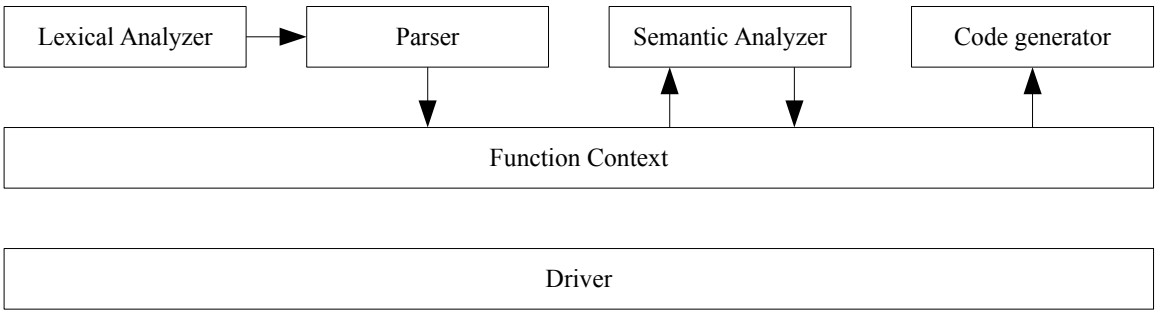


Figure 5.1.: High-level design of ImpalaJIT.

at the lexical analysis and ending with the emitted machine code. It holds references to the compiler phases objects and ensures a successively execution. The function context acts as container for all during compilation generated information. The lexical analyzer passes the scanned tokens directly to the parser. The parser creates the AST which is hold by the function context. The semantical analyzer evaluates and modifies the AST and retrieves additional information, like variable or argument names and positions. The code generator traverses the AST generates as well as emits the machine code instructions.

5.2. Configuration

ImpalaJIT relies on external input data in order to serve its purpose. The implementation of the functions which should be JIT compiled must be specified. In the following, these functions are called impala functions. ImpalaJIT supports three options to specify them which cannot be mixed.

1. Specification of an environment variable containing a configuration file path.
2. Direct passing of a configuration file path to the ImpalaJIT interface.
3. Direct definition of the impala functions in a string array and passing to the ImpalaJIT interface.

The first two options allow external function definitions. The concept is visualized in Figure 5.2. A specified configuration file contains the pathes to the actual impala function

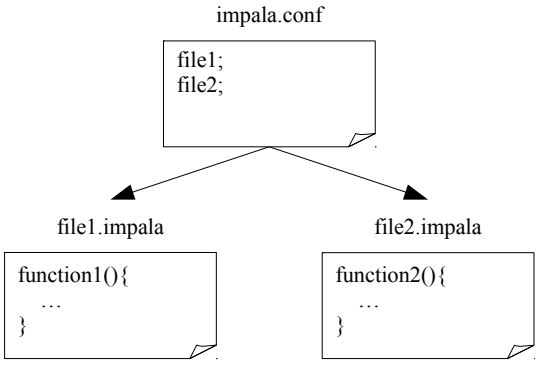


Figure 5.2.: Configuration of ImpalaJIT.

implementations. Its path can be passed to ImpalaJIT either by an environment variable, called *IMPALAJIT_CONFIG_PATH*, or by hard-coded specification through the interface. The third option allows the definition of the impala functions directly in the application code as a string array. This option is intended for development purposes.

5.3. Interface

5.3.1. Design

ImpalaJIT will be integrated in simulation applications as a library and must, hence, define an interface. A powerful interface ensures flexibility and extendablity while being easy to use. Listing 5.1 introduces the designed interface. It specifies three options for the creation of a handle which directly maps to the three configuration options introduced

Listing 5.1: Interface Design of ImpalaJIT

```
1 impalajit* create();
2 impalajit* create_with_config(const char* config_file_path);
3 impalajit* create_with_function_definitions(char** function_definitions);
4 error compile(impalajit* handle);
5 dasm_gen_func get_function(impalajit* handle, const char* function_name);
6 void close(impalajit* handle);
```

in section 5.2. The compile process is a dedicated function call and compiles all specified impala functions. This seperation allows the future extension of the interface between the configuration and compile phase. Afterwards, the function pointer to each compiled function can be retrieved by the impala function name. *dasm_gen_func* is an alias for a function interface returning one double and accepting arbitrary arguments. It is important to mention that ImpalaJIT can only handle 64 bit floating point numbers as arguments. Created resources can be released by calling *close*. The formerly retrieved function pointers still keep valid.

5.3.2. Supported Languages

Even if ImpalaJIT is written in C++, it can also be integrated in C and Fortran applications. Basically all interfaces provide the same functionality. Their actual implementation deviates according to the language characteristics. In C++ the handle is not passed as a function argument but it is possible to evaluate the pointer, for example *handle->compile()*.

In order to provide an interface for Fortran, the *iso_c_binding* module is used which excludes Fortran-77 support. In contrast to the other interfaces, the retrieved function pointers cannot be used directly. Even though the *iso_c_binding* module specifies a C function pointer type, it prior has to be mapped to a Fortran procedure pointer. A fortran procedure pointer can either be directly point to a procedure or its shape can be defined through an interface definition. An explicit interface implementation is not required, it can also be abstract, like Listing 5.2 shows. In order to call an impala function in Fortran, its abstract interface must be specified in the application and attached to a Fotran procedure pointer, shown in line 10. ImpalaJIT returns a C function pointer. Its type is shown in line 9. In order to associate a fortan procedure pointer with the target of a C pointer, the fortran routine *c_f_procpointer*, shown in line 12, can be used. Afterwards, the procedure pointer can be called.

Listing 5.2: Association of a Fortran procedure pointer with the target of a C function pointer generated by ImpalaJIT.

```
1  abstract interface
2     function impala_fun_template (a1, a2) bind(c)
3         use, intrinsic :: iso_c_binding
4         real(c_double), intent(in), value :: a1, a2
5         real(c_double) :: impala_fun_template
6     end function impala_fun_template
7  end interface
8
9  type( c_funptr ) :: cfp
10 procedure(impala_fun_template), pointer :: fpp
11
12 CALL c_f_procpointer(cfp, fpp)
```

5.4. Lexical Analyzer

The actual lexical analyzer is generated by Flex. Its specification consists of regular expressions and the corresponding tokens. In the following, the known character sequences and their categories are introduced.

- Numbers: In comparison to C, numbers are always recognized as 64 bit floating point values.
- Operators: + - * / =
- Compare Operatores: < > <= >= != ==
- Boolean Operators: && ||
- Keywords: return if else
- Identifiers: Strings starting with a letter are recognized as identifier.
- Symbols: ( ) { } , ;

In addition, it removes white spaces and newline characters since they have no special meaning in an impala function. The recognized tokens are directly passed to the parser using a buffer. The actual token declaration is also done by the parser since they are just announced to the lexical analyzer.

Listing 5.3: Grammar rule for assignments.

```
1 assignment : IDENTIFIER '=' expr
2              {
3                  $$ = new AssignmentNode(*$1, $3);
4              }
```

5.5. Parser

The parser is generated by GNU Bison. Its primary purpose is the creation of the AST. The rules section of the grammar file contains also actions if a rule is applied. Listing 5.3 shows the grammar rule for assignments as an example. If the rule can be applied, an assignment node is created, taking the expression as child and the variable name as member variable. The different parts of the matched rule can be accessed by an *$* sign followed by a number. In the example, *$1* means the *IDENTIFIER* teminal and *$3* means the *expr* nonterminal symbol. Since the *IDENTIFIER* token is a terminal symbol, its type is known. The *expr* is a nonterminal symbol which can have various types. In order to enable the tree building, all node types are inherited from a base class. The constructor of every node claims the necessary arguments in order to fully specify it. In case of an assignment, this is a String specifying the variable name and an arbritrary node which defines the value. However, the grammar rule limits the node type to an expression. Table B.1 in Appendix B lists all available node types sorted by category.

5.6. Semantic Analyzer

The semantic analyzer performs postorder traversal on the AST and gathers information which must be present before the actual code generation can start. In particular, local variables are announced. Their identifiers are pushed to a vector hold by the function context. The semantic analyzer also checks if an accessed variable was prior defined. If any error is found, a runtime error will be thrown.

Another task is the substitution of negations carrying only one constant. According to the grammar, a negative constant is represented as a negation with a constant child. This would later cause an external function call in order to negate the constant rather than direct loading of it. If such a subtree is found, it is replaced by one negative constant.

5.7. Code Generator

The code generator also performs a postorder traversal on the AST. It translates each node into machine code instructions using DynASM. Even though ImpalaJIT does not generate

Listing 5.4: Function prologue done by ImpalaJIT

```
1 | push rbp
2 | push r12
3 | mov rbp, rsp
```

intermediate code, the code generation phase is seperated into two parts. Figure 5.3 shows the design in detail. The actual DynASM instructions are hidden behind an interface in

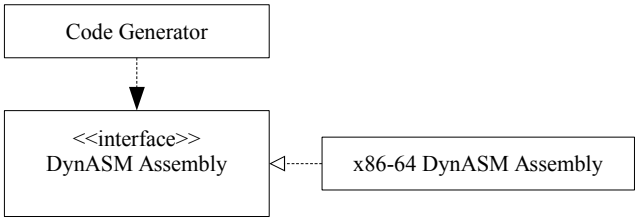


Figure 5.3.: Abstraction of DynASM instructions.

order to provide exchangeability. ImpalaJIT only supports the x86-64 instruction set but it is easily possible to add others by providing an interface implementation.

5.7.1. Function Prologue and Epilogue

A function has a frame on the run-time stack. To define the position of the frame, the register RBP is used. It holds the so called base pointer. This value is the memory address where the frame starts. Further, it grows downwards. The RSP register holds a value called frame pointer. This value is the memory address of the top of the stack. It is important to mention that the top of the stack has a lower memory address than the base pointer since the stack grows downwards. If a value is pushed onto the stack, the framepointer is decreased by eight but the base pointer stays the same. The reduction of eight happens in order to reserve space for a 64 bit value. [15]

When a function is called, the pointers are not modified. The called function has to perform the so called prologue. Listing 5.4 shows the prologue implemented by ImpalaJIT. These are always the first instructions which are executed. The current base pointer and register R12 are pushed onto the stack. Afterwards, the current frame pointer becomes the new base pointer.

Once a function is finished, it has to restore the original pointers in order to release the memory and to allow the caller to correctly proceed. These actions are called the function epilogue. Listing 5.5 shows the generated function epilogue by ImpalaJIT. It is exactly the opposit of the prologue. The old frame pointer is restored by copying the value in RBP to

Listing 5.5: Function epilogue done by ImpalaJIT

```
1 | mov rsp, rbp
2 | pop r12
3 | pop rbp
```

RSP. Afterwards, the two formely pushed register values are popped and stored in the original registers.

5.7.2. Local Variables

The semantic analyzer traverses the tree and stores the names and positions of local variables in the function context. After the prologue, a function will reserve space on the stack frame for its local variables. This is achieved by decreasing the frame pointer by the eightfold amount of local variables since a variable has a size of eight byte. Figure 5.4 shows the stack frame after the space reservation. When the code generator traverses an assignment or variable node, a position lookup is performed. A local variable can then be accessed by subtracting the eightful amount of its index from the base pointer.

5.7.3. Function Arguments

ImpalaJIT compiled functions support any amount of 64 bit floating point values as arguments. According to the System V Application Binary Interface [15], the first eight 64 bit floating point arguments are passed in registers XMM0-XMM7. More are passed via the run-time stack. Figure 5.4 shows where function arguments are placed on the stack. The calling function places them on top of the stack before it invokes the call. Considering the called function, an argument has a negative stack position. Functions compiled by ImpalaJIT are able to deal with both passing methods. The semantic analyzer stores the name and the position of the argument in the function context. When a code generator traverses an argument node, a position lookup is performed. Depending on the position, the right register or memory address is accessed.

5.7.4. Expressions

ImpalaJIT makes use of the Reverse Polish Notation (RPN) in order to evaluate expressions. In comparison to the Infix Notation, the operands are written before the operator. For example, the expression *5 * 8 - 4* would be stated as *5 8 * 4 -* in RPN. This notation fits perfect for a stack-based evaluation of expressions. The operands are pushed onto a stack. When the next symbol is an operator, all items are popped and the operator is applied. The result of the operation is again pushed onto the stack.

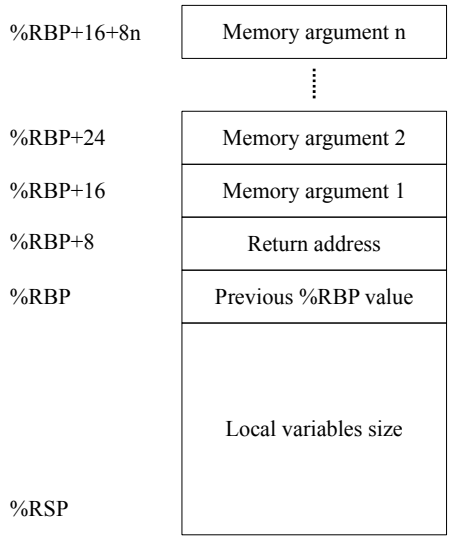


Figure 5.4.: Stack frame layout according to System V Application Binary Interface. [15]

The transition of the Infix Notation to the RPN is implicitly done by evaluating the AST. ImpalaJIT performs postorder traversal. This strategy returns first the operands since they are the children. Next, the operator, which is the parent, is returned. [16]

Following this concept, ImpalaJIT needs to hold a stack where the operands are stored. To avoid confusion with the rune-time stack, it is further called RPN stack. Since only 64 bit floating point values are supported by ImpalaJIT, the sixteen SSE registers, XXM0-XMM15, are used for stack building. If arguments are passed to the function, the first unused register represents the first stack position instead of XMM0. If more elements than available registers should be stored, the stack is continued in memory. In detail, the stack frame is extended downwards.

This design meshes up with assembly. The instructions for the basic arithmetical operations require a register or a memory location as source and a register as destination. The values stored in these two locations are the operands. The result of the calculation is stored in the destination register. [17]. As long as the RPN stack fits into registers, an operation can be performed by one instruction. If the RPN stack has reached the memory, nevertheless, a register must be used. This causes more instructions since a register must be saved and restored.

5.7.5. If/Else Statements

ImpalaJIT supports If statements and If-Else statements. On assembly level such statements can be realized by compare and jump instructions and jump targets, called labels.

Exemplary, Listing 5.6 shows the comparison of two 64 bit floating point numbers and

Listing 5.6: If-Else statements as DynASM instructions

```
1 | cmpsd xmm1, xmm0, operator // Compare XMM1 with XMM0
2 | ptest xmm1, xmm1 // Set the zero flag if XMM0 is zero
3 | jz 1 // Jump to label 1 if the zero flag is set
4 [...] // If-body
5 | jmp 2 // Jump to label 2
6 | 1: // Label 1
7 [...] // Else-body
8 | 2: // Label 2
```

the subsequent jump instructions. In line 1, the value stored in XMM1 is compared with the value in XMM0. It sets all zeros in XMM1 if the condition is wrong, all ones if its true. The *PTEST* instruction checks if XMM1 contains only zeros, if true, the zero flag is set. In line 3, a conditional jump to label 1, which marks the Else-body, is performed. The condition for the jump is a set zero flag. Consequently, the subsequent instructions constitutes the If-body. At the end of the If-body, an unconditional jump to label 2 is performed in order to skip the Else-body.

Impala functions first evaluate the two operands of the condition since they can also be expressions. Afterwards, the comparison is perfomed and the RPN stack is popped two times because the operands are not needed anymore. ImpalaJIT also supports the boolean operators *&&* and `||` in conditions. In order to compile a *&&* junction, the conditions of the junction are subsequently evaluated. If one of them evaluates to false, a jump to the Else-block is performed. In order to compile a `||` junction, the jump logic must be inverted. A label is placed at the beginning of the if-body and a jump is performed if one of the conditions evaluates to true.

5.7.6. Function Calls

ImpalaJIT generated functions are capable of calling other functions. The requirements such a function must meet in order to be callable are listed in the following:

- It can take zero up to eight 64 bit floating point arguments, but not more. ImpalaJIT supports only arguments passing in registers.
- It has to return a 64 bit floating point value.

Most of the functions provided by the C Standard Math library can be called by impala functions because they deal with floating point values. Since C++ does not support reflection, ImpalaJIT holds a map of function names and the corresponding pointers. This map can be safely extended as long as the functions meet the requirements.

Considering the run-time stack, a called function gets space on the top of it. An impala function executes the following steps in order to call another function with n arguments.

1. Backup the RPN stack and passed arguments since the XMM registers are volatile. The register values are pushed onto the run-time stack. The arguments for the to be called function are currently the top n RPN stack items. These items are skipped.
2. The arguments of the to be called function are moved from the top of the RPN stack to the passing registers XMM0 to XMM(n).
3. Backup the current frame pointer in register R12. This register is non-volatile which means the called function is in charge of restoring its value.
4. Since step 1 did not include the modification of the frame pointer, the eightfould amount of stored RPN stack items is subtracted from it.
5. The System V ABI expects a run-time stack alignment of 16 bit. This means, the frame pointer is further decreased until it is aligned on 16 bit.
6. The actual address of the to be called function is moved to the RAX register.
7. The function call, using the value of the RAX register as source, is performed.
8. The RSP register is restored by moving the value of R12 to RSP.
9. The RPN stack is popped n times in order to remove argument values of the called function. The function result, passed in XMM0 register, is pushed onto the RPN stack.
10. The RPN stack and originally passed arguments are restored.

6. Results

This chapter presents the results of the ImpalaJIT integration in SWE and SeisSol. In addition, the results of a performance benchmark, which constitutes a bad-case scenario for ImpalaJIT, are presented. In order to validate if ImpalaJIT performed proper calculations, the simulation results were compared with references. The reference results were obtained by performing the same simulation but with Ahead-of-time compiled scenarios. In order to evaluate the performance, time measurements were taken in all testruns.

All tests were performed on the following system:

- Linux Cluster CoolMUC2
  Intel Xeon E5-2690 v3 ("Haswell"), 28 nodes per core, 64 GB RAM per node

6.1. Pythagoras Benchmark

The primary advantage of an Ahead-of-time compiler, in comparison to ImpalaJIT, is vectorization. In order to produce a bad-case scenario for ImpalaJIT, an impala function was executed in a fully vectorizable loop. Since Ahead-of-time compilers perform sophisticated optimizations, a worst-case scenario is hard to find. Listing 6.1 shows how the JIT compiled function is called. The actual impala function is shown in Listing 6.2. Time measures were taken before and after the outer loop. As stated above, the dynamic compiled version was compared with an Ahead-of-time compiled version. This test was performed two times in order to get results using the Intel compiler and the GNU compiler. Figure 6.1 shows the measured execution times for 2.5 billion function calls. ImpalaJIT was approximately six times slower than GNU and 11 times slower than Intel. The primary reason for the measured gaps was, as already mentioned, the applied vectorization of Ahead-of-time compilers. Interestingly, when using the Intel compiler, the execution time decreased by approximately 50% compared to the GNU compiler. Also, ImpalaJIT needed less amount of time. This result can be explained by the linked math libraries. When using the Intel compiler, the linker links an optimized Intel math library. ImpalaJIT also relies on functions of this library in order to calculate the power or sqrt function.

6.2. Shallow Water Equations

SWE already defines a scenario interface by design containing five functions which facilitated the integration. Listing 6.3 shows how this interface was used to JIT compile

Listing 6.1: ImpalaJIT function call in pythagoras benchmark

```
1  int main(int argc, char* argv[]){
2      impalajit::Compiler compiler(CONFIG_FILE_PATH);
3      compiler.compile();
4      dasm_gen_func pythagoras = compiler.getFunction("pythagoras");
5      for(int i = 0; i<LOOP_1_BOUNDARY; i++) {
6          for (int j = 0; j < LOOP_2_BOUNDARY; j++) {
7              pythagoras(static_cast<double>(i), static_cast<double>(j));
8          }
9      }
10 }
```

Listing 6.2: Impala function calculating the hypotenuse of a right triangle

```
1  pythagoras(a, b){
2      return sqrt(pow(a,2)+pow(b,2));
3  }
```

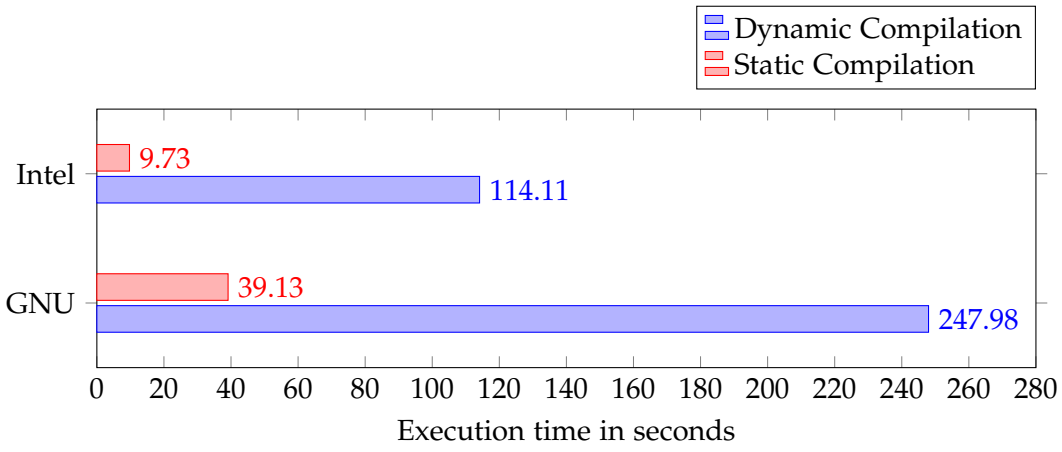


Figure 6.1.: Execution times of the pythagoras benchmark. The pythagoras function was invoked 2.5 billion times.

the *getWaterHeight* and *getBathymetry* function as an example. The actual implementation of the two functions is indicated in Listing 6.4. For the testruns, the radial dam break scenario was chosen.

Listing 6.3: Integration of ImpalaJIT in SWE

```
1  impalajit::Compiler compiler("impala.conf");
2  dasm_gen_func i_getWaterHeight;
3  dasm_gen_func i_getBathymetry;
4
5  SWE_Scenario::SWE_Scenario(){
6      compiler.compile();
7      i_getWaterHeight = compiler.getFunction("getWaterHeight");
8      i_getBathymetry = compiler.getFunction("getBathymetry");
9  }
10
11 float SWE_Scenario::getWaterHeight(float x, float y){
12     return i_getWaterHeight(x, y);
13 }
14
15 float SWE_Scenario::getBathymetry(float x, float y) {
16     return i_getBathymetry(x, y);
17 }
```

Listing 6.4: Impala functions of the radial dam break scenario in SWE

```
1  getWaterHeight(x, y){
2      if( sqrt( (x-500)*(x-500) + (y-500)*(y-500) ) < 100 ) {
3          return 15;
4      }
5      else{
6          return 10;
7      }
8  }
9
10 getBathymetry(x, y){
11    return 0;
12 }
```

In sum, it is possible to replace all currently implemented scenarios by impala functions, except of the Asagi scenario since it calls external functions. The initialization times of the radial dam break scenario, when using a grid size of 4000x4000, are shown in Figure 6.2. The simulation was executed at one node and it was not parallelized.

The initialization time of the bathymetry shows that ImpalaJIT consumes around 20% more time for the returning of a constant in comparison to the Intel compiler. Considering the *getWaterHeight* function, which contains an if/else statement and relies on arguments, it required around twice the amount of time. ImpalaJIT increased the overall initialization time by roughly 36%.

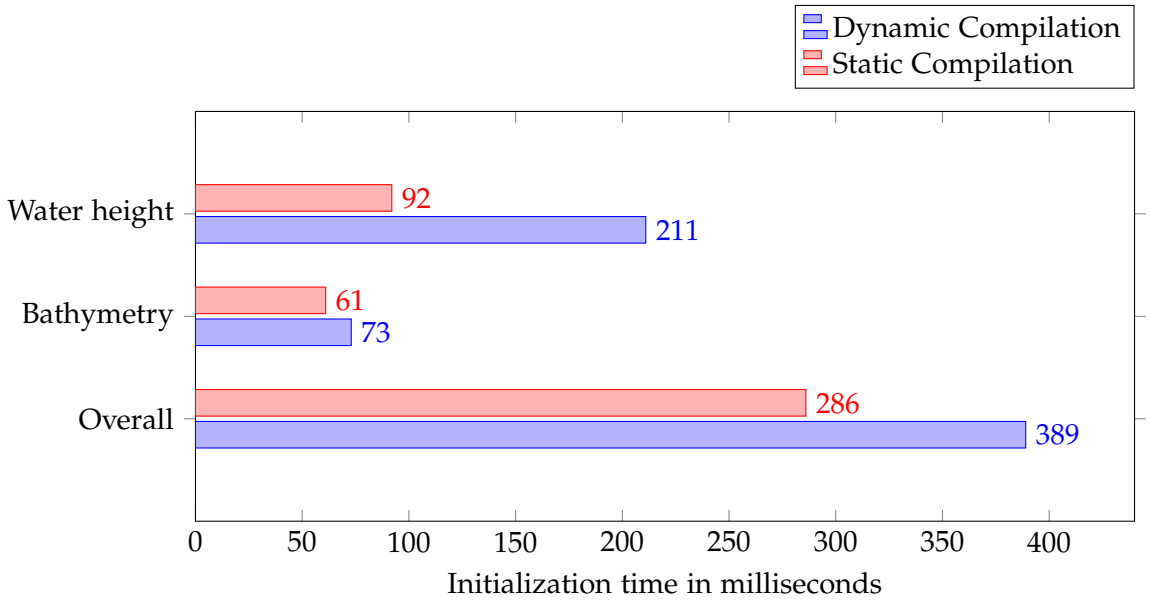


Figure 6.2.: Initialization time of the radial dam break scenario in SWE seperated by parameters. A grid size of 4000x4000 was used for the simulation.

The actual simulation outputs were compared in order to validate if the JIT compiled functions returned the same values as their Ahead-of-time compiled counterparts. The generated output files fully accord which means ImpalaJIT compiled the functions properly. Figure 6.3 shows the visualized simulation output at three points in time for a second testrun with a grid size of 1000x1000. The first row visualizes the output when JIT compilation is applied. In the second row, the reference output for the same points in time is displayed.

6.3. SeisSol

SeisSol does not define an uniform scenario interface which complicated the ImpalaJIT integration. A huge Select Case compares an input parameter with hard-coded reference values and cares for the right scenario selection. This initializing strategy was kept in order to also support not yet migrated scenarios. Most of the case statements of migrated scenarios can be combined. Following this pattern, the amount of diverse cases will

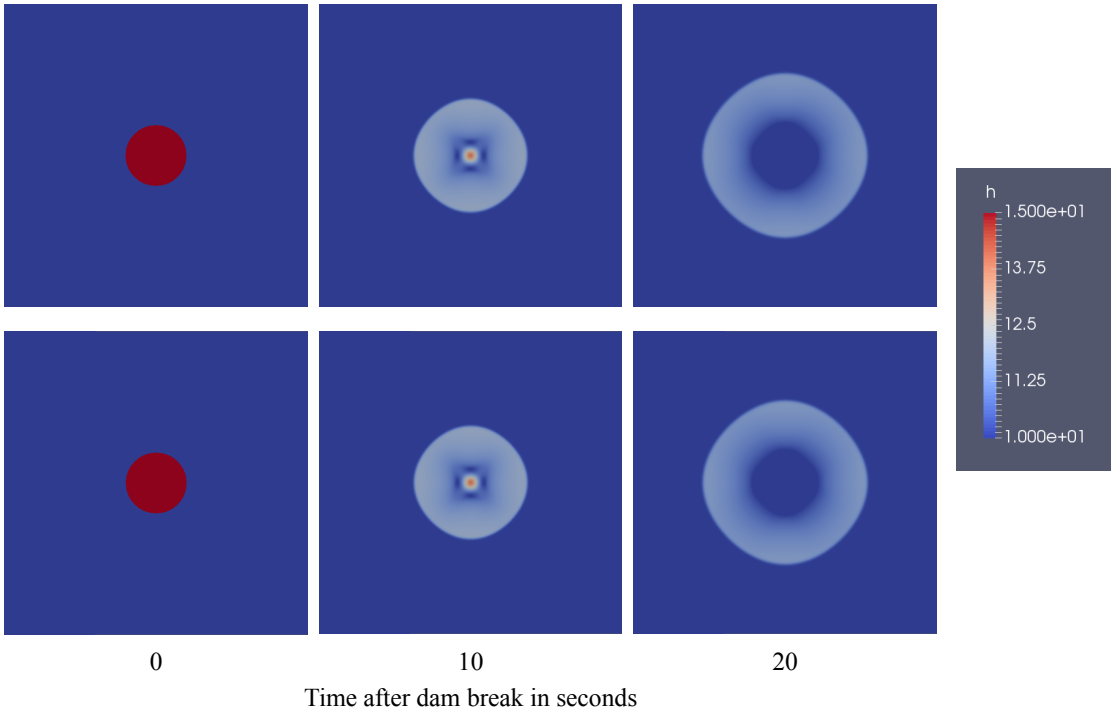


Figure 6.3.: Simulation output of SWE using the radial dam break scenario at three points in time. The first row shows the output when ImpalaJIT was used for data initialization. The second row shows the original output.

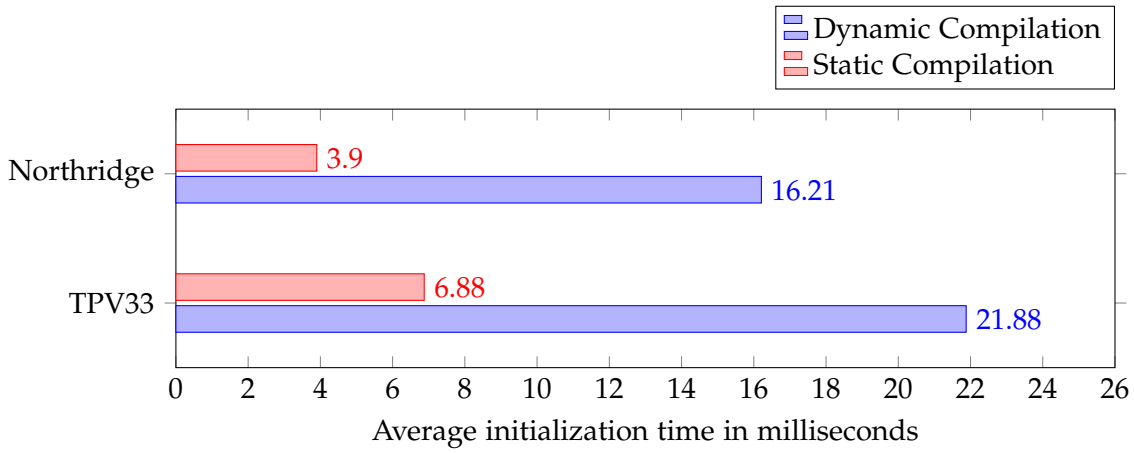


Figure 6.4.: Average initialization time of the northridge and TPV33 scenario of SeisSol per process.

decrease with no impact on other parts of SeisSol. In order to not further rely on the input parameter for migrated scenarios, it is reasonable to set the combined case as default.

Listing 6.5 shows how SeisSol's TPV33 scenario was migrated. Lines 28 to 30 show the calls of the JIT compiled function. In line 27 a coordinate transformation is performed which is still kept in the application in order to focus on the actual scenario. Exemplary, Listing 6.6 indicates the implementation of the TPV33 case as an impala function. Overall, two scenarios, Northridge and TPV33, were migrated and one testrun for each scenario was performed.

The first testrun used the TPV33 scenario. SeisSol was executed on 8 nodes, parallelized by 8 MPI processes and 28 threads per process. The second testrun used the northridge scenario and was executed on 32 nodes, parallelized by 32 MPI processes and 28 threads. It is important to mention that the data initialization of each process was performed by only one thread even though an impala function is thread safe. The execution time of the loop was measured. Figure 6.4 shows the average initialization time per process of both testruns. Considering the TPV33 scenario, ImpalaJIT claimed around triple the amount of time compared to the Intel compiler. For the northridge scenario, ImpalaJIT was around four times slower.

Listing 6.5: Integration of ImpalaJIT in SeisSol

```
1  SUBROUTINE ini_MODEL(MaterialVal,OptionalFields,EQN,MESH,IC,IO,DISC,BND)
2  [...]
3      USE impalajit
4      USE, intrinsic :: iso_c_binding
5      IMPLICIT NONE
6          ABSTRACT INTERFACE
7                  FUNCTION impala_fun_template (a1, a2) BIND(c)
8                          USE, INTRINSIC :: ISO_C_BINDING
9                          REAL(C_DOUBLE), INTENT(IN), VALUE :: a1, a2
10                         REAL(C_DOUBLE)    :: impala_fun_template
11                 END FUNCTION impala_fun_template
12         END INTERFACE
13 [...]
14     TYPE( c_ptr ) :: handle
15     TYPE( c_funptr ) :: cfp
16     PROCEDURE(impala_fun_template), pointer :: fpp
17 [...]
18     handle = impalajit_compiler_create_with_config("impala.conf")
19     CALL impalajit_compiler_compile(handle)
20     cfp = impalajit_compiler_get_function(handle, "getMaterialVal")
21     CALL c_f_procpointer(cfp, fpp)
22 [...]
23     SELECT CASE(EQN%LinType)
24 [...]
25     CASE(33)
26         DO iElem = 1, MESH%nElem
27             y = MESH%ELEM%xyBary(2,iElem) !average y inside an element
28             MaterialVal(iElem,1)=fpp(y, 1.0d0)
29             MaterialVal(iElem,2)=fpp(y, 2.0d0)
30             MaterialVal(iElem,3)=fpp(y, 3.0d0)
31 [...]
```

Listing 6.6: Impala function of the TPV33 scenario in SeisSol

```
getMaterialVal(y, i){
        if(i == 1){
                return 2670;
        }
        if(y < -800){
                if(i==2){
                        return 2.8167175680;
                }
                if(i==3){
                        return 28176157560;
                }
        }
        if(y >= -800 && y <= 800){
                if(i==2){
                        return 12514890750;
                }
                else if(i==3){
                        return 12517093500;
                }
        }
        if(y > 800){
                if(i==2){
                        return 32038120320;
                }
                if(i==3){
                        return 32043759360;
                }
        }
}
```

7. Discussion

7.1. Implications

Considering the requirements defined in chapter 2, the majority of them could be met. The three points not implemented so far are array support, vectorization and the integration of ASAGI. The pythagoras benchmark violates the performance requirement when the Intel compiler is used. However, it constitutes a bad-case scenario for ImpalaJIT. The gap in order to meet the requirement was around 17%.

The achieved results showed that ImpalaJIT is capable of increasing the flexibility of data initialization of simulation applications. The universal interface design and its availablity for the most common languages in High Performance Computing fosters the integration in existing applications. Considering SWE, all scenarios can be migrated to impala functions, except of the ASAGI scenario. In SeisSol, nine of the 25 cases can be implemented as impala functions. 16 cases contain an array access in order to initialize the data. This functionality is, at the current state of ImpalaJIT, not supported by impala functions. However, it acts as an starting point ready to get extended in various directions.

The time measures indicated that using ImpalaJIT causes a performance loss in comparison to the utilization of Ahead-of-time compilers. Nonetheless, it has succeeded providing a solution which enhances the flexibility of data initialization. The pythagoras benchmark indicated that in bad cases impala functions are eleven times slower than Ahead-of-time compiled code. It depends on the targets of an application if this drawback is acceptable.

Both applications, SWE and SeisSol, are based on static grids. In applications, utilizing dynamic adaptive mesh refinement, the impala functions would be also called at simuation time. For such applications, the performance drawback, considering the overall time, would increase and may lead to unacceptable simulation times. To consitute a possibility for these applications, ImpalaJIT needs to be extended by vectorization.

7.2. Future Work

Currenty, ImpalaJIT is capable of replacing nine SeisSol scenarios. 16 scenarios access data structures like arrays and, hence, are not able to be migrated. In order to increase the number of supported scenarios, ImpalaJIT must be extended to pointer support. This includes function pointers as well as pointers to data structures.

The results also showed that using impala functions causes a performance loss. This is primarily caused by missing vectorization capabilities. Currently, the loop containing the impala function call is Ahead-of-time compiled. Such a design prohibits vectorization. One way of enabling it is the integration of the loop in the impala function. An optimization phase could analyze the loop and vectorize it if possible. Another approach is a developer supported vectorization. The impala function could still be called in an Ahead-of-time compiled loop but operate on vectors identified by passed pointers. The function is then applied to various vector elements rather than one single element. It would also have to return a vector. This would decrease the overall function calls.

At the moment, ImpalaJIT supports only 64 bit floating point values as arguments and return value. Supporting various types is crucial in view of functionality. For example, in order to access an array by index, integer types are required. Considering vectorization, 32 bit floating point numbers could also increase the performance.

ASAGI, another library targeting the data initialization of simulation applications was introduced. It is primarily utilized if input data is available in a discretized form. In comparison to scenario definitions by continuous functions, it is not guaranteed that a desired data point is always specified. ImpalaJIT could be used for the provision of JIT compiled interpolations.

8. Summary

This thesis indicated how data initialization of simulation application can benefit from JIT compiling.

Two applications, SWE and SeisSol, were analyzed in order to introduce current scenario implementations. A developed solution should be universal applicable. Application developers were interviewed in order to elicitate requirements. The most important findings were, beside the functional requirements, the independence of other libraries and the performance requirement.

Afterwards, several approaches enabling JIT compiling were analyzed. It turned out that only few software libraries exist supporting this functionality. Constructing a compiler customized for the utilization in simulation applications was assessed as the most convenient solution.

Furthermore, the underlying concepts of the developed library were explained. The overall design is oriented to the four compiler phases, lexical analysis, syntax analysis, semantic analysis and code generation. The compiler is able to read source files in a C-like syntax and translating them to machine code at runtime. A simple interface, available for C, C++ and Fortran, ensures the easy integration in existing simulation applications.

The results showed how this interface was used to integrate ImpalaJIT in SWE and SeisSol. SWE currently contains six scenarios. Five of them can be fully replaced by impala functions. Only one special scenario relying on an external library is not able to be migrated. SeisSol consists of 25 scenarios. A strategy how current implementations can be kept while new scenarios are implemented as impala functions was introduced. ImpalaJIT can cover nine of the SeisSol scenarios. Time measures verified the aptitude of ImpalaJIT for data initialization.

This thesis showed that it is possible to utilize JIT compilation in simulation applications in order to improve the flexibility of data intialization.

A. Interview Guidelines

Einleitung
Der Interviewer stellt kurz JIT Kompilierung vor. Im weiteren wird kurz auf den Einsatz von JIT in Simulationsanwendungen eingegangen. Danach wird der Zweck des Interviews erleutert.

- Das Interview dient zur Anforderungserhebung. Die zu entwickelnde Software soll eine möglichst einfache Schnittstelle für Anwender bereitstellen. Damit dies gewährleistet werden kann, müssen die Anforderungen der Anwender erhoben und miteinbezogen werden.

Es folgt die Frage ob das Interview aufgezeichnet werden darf und eine Datenschutzerklärung.

- Die Audioaufzeichnungen dienen zur Nachbearbeitung der Interviews und werden nicht an Dritte weitergegeben.
- Die im Interview erhobenen Daten werden strukturiert und in Anforderungen überführt. Diese Anforderungen werden in der Masterarbeit dokumentiert. Ebenso behalte ich mir vor die erhobenen Daten in jeglicher anderer Form für die Masterarbeit zu verwenden.
- Eine Verwendung der Daten für andere Zwecke ist ausgeschlossen.

Fragenteil
Die Fragen werden nicht abgelesen, sondern im Gespräch „erarbeitet“. Codesichtung ist explizit erwünscht. Die Aufnahmezeit und der Dateiname werden dabei notiert, damit später der gesichtete Code mit gesprochenen Text nachvollzogen werden kann. Fragen zu Eingabedaten:

- Wie werden in der Anwendung Szenarien beschrieben?
- Welchem Schema folgen die Eingabedaten/Funktionen?
- Können Beispielszenarien für die Masterarbeit bereitgestellt werden?

Fragen zu Szenarioimplementierung:

- Werden Werte an Gitterpunkten berechnet?
- Wie sind diese Berechnungen momentan implementiert?
- Welche mathematischen Operationen werden dafür typischerweise verwendet?
- Welche mathematischen Operationen werden häufig verwendet und nicht durch Standard C library unterstützt?
- Welche Datentypen können als Operanden auftreten?
- Kommen für die Berechnung rekursive Funktionen oder verschachtelte Funktionen zum Einsatz?
- Verwenden die Funktionen Pointer?
- Könnnen die Berechnungen vektorisiert werden?

Abschluss

- Wie ist die generelle Meinung zu dem Projekt?
- Gibt es relevante Punkte, welche bis jetzt nicht zur Sprache gekommen sind?

B. Node Types of the Abstract Syntax Tree

Table B.1.: Available AST node types sorted by category .

Category	**Node type**	**Example**
expression	ConstantNode	5;
expression	VariableNode	x;
expression	NegationNode	-(5+2);
expression	AdditionNode	3+3;
expression	SubtractionNode	3-3;
expression	MultiplicationNode	3*3;
expression	DivisionNode	3/3;
expression	ExternalFunctionNode	sqrt(2);
assignment	AssignmentNode	a=3;
comparison	CompareNode	a==b
boolean junction	BooleanAndNode	a==b && c==d
boolean junction	BooleanOrNode	a==b || c==d
if / else	IfStmtNode	if(a==b){c=3;}
if / else	IfElseStmtNode	if(a==b){c=3;} else{c=4;}
wrapper	IfBodyNode	expressions, assignments, return
wrapper	ElseBodyNode	expressions, assignments, return
return	ReturnNode	return x;

List of Figures

List of Tables

Bibliography

[1] The SeisSol Team. (2015). Official seissol github repository, [Online]. Available: https://github.com/SeisSol/SeisSol (visited on 02/25/2017).

[2] A. Breuer and M. Bader, "Teaching parallel programming models on a shallow-water code.," in *ISPDC*, M. Bader, H.-J. Bungartz, D. Grigoras, M. Mehl, R.-P. Mundani, and R. Potolea, Eds., IEEE Computer Society, 2012, pp. 301–308, ISBN: 978-1-4673-2599-8.

[3] The SCCS Team. (2012). Official swe github repository, [Online]. Available: https://github.com/TUM-I5/SWE (visited on 01/09/2017).

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN: 0-201-63361-2.

[5] C. Pelties, A. Gabriel, L. Passone, A. Breuer, S. Rettenberger, and A. Atanasov, "Seissol: The ader-dg method for seismic wave propagation and earthquake rupture dynamics," in *SCEC Annual Meeting 8-11 September 2013, Palm Springs, California, USA*, Sep. 2013.

[6] S. Rettenberger, O. Meister, M. Bader, and A.-A. Gabriel, "Asagi: A parallel server for adaptive geoinformation," in *Proceedings of the Exascale Applications and Software Conference 2016*, ser. EASC '16, Stockholm, Sweden: ACM, 2016, 2:1–2:9, ISBN: 978-1-4503-4122-6. DOI: 10.1145/2938615.2938618.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN: 0-201-10088-6.

[8] J. Aycock, "A brief history of just-in-time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003, ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[9] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, and A. Iyer, "Towards green aviation with python at petascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 1:1–1:11, ISBN: 978-1-4673-8815-3.

[10] V. Paxson. (1987). Flex, a fast scanner generator, [Online]. Available: https://github.com/westes/flex (visited on 03/06/2017).

[11] Free Software Foundation, Inc. (2014). Gnu bison, [Online]. Available: https://www.gnu.org/software/bison (visited on 03/06/2017).

[12] C. Lattner. (2003). The llvm compiler infrastructure project, [Online]. Available: http://llvm.org (visited on 03/04/2017).

[13] M. Pall. (2005). The luajit project, [Online]. Available: https://luajit.org (visited on 03/04/2017).

[14] F. Bellard. (2001). Tiny c compiler, [Online]. Available: http://www.bellard.org/tcc (visited on 02/01/2017).

[15] Intel Cooperation. (2013). System v application binary interface, [Online]. Available: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf (visited on 02/19/2017).

[16] C. L. Hamblin, "Translation to and from polish notation," *The Computer Journal*, vol. 5, no. 3, p. 210, 1962. DOI: 10.1093/comjnl/5.3.210. eprint: /oup/backfile/Content_public/Journal/comjnl/5/3/10.1093/comjnl/5.3.210/2/5-3-210.pdf.

[17] Intel Cooperation. (2016). Intel® sse4 programming reference, [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (visited on 02/19/2017).