Automation of manual code optimization via DSL-directed AST-manipulation

Automatisierung von manuellen Codeoptimierungen durch DSL-gesteuerte AST-Manipulation

Bachelorarbeit

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Jonas Gresens
E-Mail-Adresse:	2gresens@informatik.uni-hamburg.de
Matrikelnummer:	6419376
Studiengang:	Informatik
Erstgutachter:	Dr. Julian Kunkel
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Dr. Julian Kunkel

Hamburg, den 27.06.2016

Abstract

Program optimization is a crucial step in the development of performance critical applications but in most cases only manually realizable due to its complexity. The substantial structural changes to the source code reduce the readability and maintainability and complicate the ongoing development of the applications. The objective of this thesis is to examine the advantages and disadvantages of an AST-based solution to the conflicting relationship between performance and structural code quality of a program. For this purpose a prototype is developed to automate usually manual optimizations based on instructions by the user. The thesis covers the design and implementation as well as the evaluation of the prototype for the usage as a tool in software development. As a result this thesis shows the categorical usability of the AST-based approach and the need for further investigation.

Inhaltsverzeichnis

1 Introduction **5**
1.1 Software Quality . . . 5
1.2 Motivation . . . 5
1.2.1 AIMES . . . 6
1.3 Goals of this Thesis . . . 7
1.4 Outline . . . 7

2 Background **9**
2.1 Abstract Syntax Tree . . . 9
2.2 Parsing . . . 9
2.3 Optimization . . . 9
2.3.1 Function Inlining . . . 10
2.3.2 Loop Fusion . . . 10
2.3.3 Loop Unswitching . . . 10

3 Related Work **11**
3.1 OpenMP . . . 11
3.2 Groovy . . . 11
3.3 GCC . . . 11
3.4 LLVM . . . 12
3.5 ROSE . . . 12

4 Design **13**
4.1 Requirements . . . 13
4.2 Functionality . . . 13
4.3 Architecture . . . 14
4.3.1 Initial Approach . . . 14
4.3.2 Revised Approach . . . 14
4.4 Possible Uses . . . 15

5 Implementation **16**
5.1 Methodology . . . 16
5.2 Architecture . . . 16
5.3 Abstract Syntax Tree . . . 17
5.3.1 Types of AST Nodes . . . 17
5.4 Parsing . . . 18
5.5 Unparsing . . . 19

5.6 C99 Subset . . . 20
5.6.1 Supported Language Features . . . 20
5.6.2 AST Manipulations . . . 21
5.6.3 DSL . . . 23

6 Evaluation **24**
6.1 Requirements . . . 24
6.1.1 Functionality . . . 24
6.1.2 Extensibility . . . 29
6.1.3 Integrability . . . 29
6.2 Design & Implementation . . . 29
6.2.1 Performance . . . 29
6.2.2 AST-based Approach . . . 30

7 Conclusion **32**
7.1 Summary . . . 32
7.2 Future Work . . . 32

Bibliography **33**

Appendices **34**

List of Figures **35**

List of Listings **36**

1 Introduction

Over the last few decades computers have become an ubiquitous part of the modern world. They are widespread and used to manage and process all kinds of data in almost every field of application such as education, administration, manufacturing, logistics and research. The main reasons for the success of computers are the speed at which they can do repetitive tasks compared to humans and their rapid adaptability for different purposes via suitable software packages.

1.1 Software Quality

As programs increase in size as well as complexity and are used to control progressively more important systems, their quality becomes increasingly more important. This leads to the current international standard for the evaluation of software quality ISO/IEC 25010:2011 (titled *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*). It defines eight distinct software product quality characteristics, which can be broken down into two categories:

- **Functional qualities** reflect how well a program complies with a given design, based on its specified functional requirements. They and are directly recognized by a user. These qualities are functional suitability, reliability, performance efficiency, usability, security and compatibility.
- **Structural qualities** refer to how well a program meets non-functional requirements. They invisible to the user but of the highest importance to a developer, like maintainability and portability.

1.2 Motivation

When developing software the first objective typically is to achieve a certain level of basic functional qualities such as functional suitability, usability, reliability, security and compatibility. After achieving these, most programs have to be optimized to meet their required execution speed and/or memory usage to be ready for use in production.

The optimizations may be done automatically by an optimizing compiler but are in some cases required to be manually done by the developer at the source code level. This is caused by the compiler not being able to perform every type of optimization, as they

may require knowledge of the input data or the quirks of the target computers hardware as well as an extensive analysis of the dependencies in the program.

But manual code optimizations also have a drawback, the changes in the source code often reduce its structural quality (i.e. maintainability and portability) and make further development more difficult as well as often worsen the general readability.

The double-edged sword nature of manual optimizations can be ignored as long as the program is not developed any further, but when changes have to be made, there are two ways of implementing them, each with its pro and cons:

- Using the optimized code as the basis for the next version spares the need to completely reoptimize the code but is usually more complex and time consuming due to possibly significant changes to the code structure.
- Further developing the unoptimized version is usually much easier but eventually also takes a lot of time, as it naturally requires a reoptimization of the program later on to achieve the same performance as the previous version.

Manual optimizations are especially important in the field of high performance computing (HPC), where programs typically work on larger problems than encountered in the world of desktop computing and also run longer (up to several months [1]), despite being already executed on powerful supercomputers. Optimizing the performance critical sections and tuning the program for one specific machine is beneficial regarding computation time, hardware utilization and power usage. It ultimately reduces the operating costs of the software and makes research both faster and cheaper. Due to this major advantage the consequent loss of structural code quality is simply accepted, although it tremendously increases the effort required to port the program to a different supercomputer or to incorporate new scientific insights.

An example for software, which is typically used in HPC and continuously gets further developed, are the computational models utilized by climate scientists to simulate the climate development.

1.2.1 AIMES

At institutions like the *Deutsches Klimarechenzentrum* and the *Universität Hamburg*, where currently research to solve this dilemma, primarily but not exclusively in the domain of climate research, takes place: The *Advanced Computation and I/O Methods for Earth-System Simulations* (AIMES) project "address[es] the key issues of programmability, computational efficiency and I/O limitations that are common in next-generation icosahedral earth-system models". [aim]

To enhance the programmability, AIMES abstracts from the details of the different dialects via the use of a domain specific language (DSL), that will enable the scientists to "express the model operators in a natural and simple way". [aim] This approach requires a broad set of tools, including a source-to-source translation tool, that provides the automatic code manipulations, necessary to convert code enriched with DSL statements into compilable code, optimized for a given architecture.

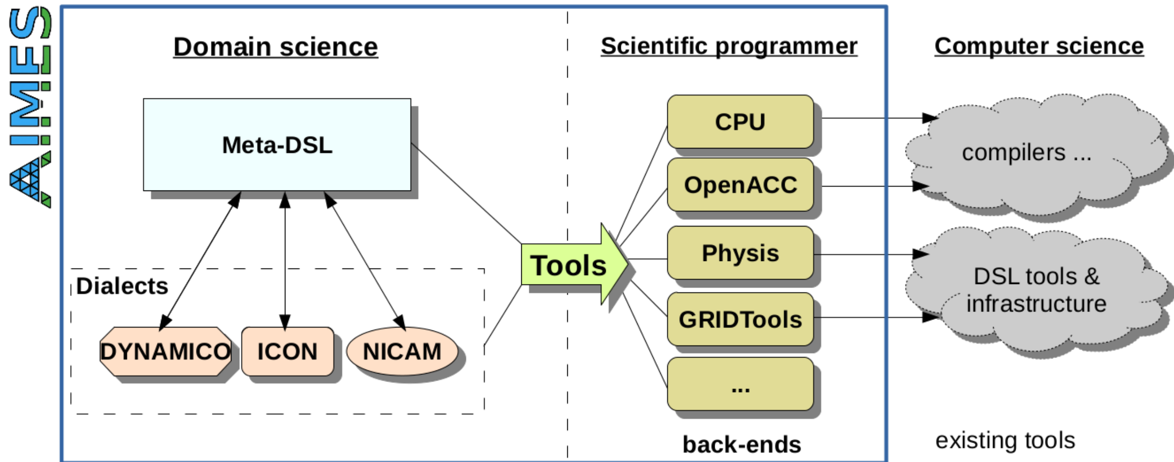

Figure 1.1: AIMES' approach to enhance programmability
[aim]

The AIMES project shares its approach, of using DSL-based abstraction and automatic code generation to ease development, with the concept of model-driven engineering (MDE), which temporarily has been a hype in software development. What makes AIMES different is its realisation, which is less radical than MDE, as it aims to extend the currently most prominent used language in the field (Fortran), instead of completely replacing it with DSLs.

The suggested design of AIMES' source-to-source translation tool uses an abstract syntax tree (AST) for the internal code representation, which enables the individual translation steps to be implemented as abstract high level tree manipulations.

1.3 Goals of this Thesis

The goal of this thesis is to explore the advantages and disadvantages of the AST-based approach for high-level code manipulations. For this purpose a prototype is developed, based on the requirements for an exemplary tool within the AIMES project, and used to evaluate the feasibility and suitability of the design. Considering the great effort and subject-specific expertise required to develop an early version of the actual translation tool, the scope of the prototype will be limited to the automation of usually manual code optimizations.

1.4 Outline

This thesis is divided into chapters as follows:

- Chapter 2, Background, describes the different generic concepts and methods used throughout the bachelor thesis;

- Chapter 3, Related Work, presents related work in the domain;
- Chapter 4, Design, illustrates the design of the prototype;
- Chapter 5, Implementation, presents its implementation and showcases the integral parts of the code;
- Chapter 6, Evaluation, analyzes the different strengths and weaknesses of the design approach;
- Chapter 7, Conclusion, summarizes the work and gives future perspectives.

2 Background

This chapter briefly explains the used concepts like abstract syntax trees, parsing, optimization in general and three specific techniques (function inlining, loop fusion and loop unswitching).

2.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree representation of the hierarchical syntactic structure of source code written in a programming language. Each node of the tree denotes a syntactical construct (structure) occurring in the source code. In general abstract syntax trees are produced by a parser and used as data structures in compilers for the translation to three-address code. [ALSU07, p. 41]

2.2 Parsing

Parsing, also called syntax analysis, is the second step in a compiler and uses the tokens, produced by a lexical analyzer, and a context-free grammar to create a tree-like representation that depicts the grammatical structure of the tokens (the AST). However, the whole process of lexical and syntactical analysis of a transforming text into a data structure is commonly referred to as parsing, the reversed action is called unparsing. [ALSU07, p. 8]

2.3 Optimization

Program optimization or software optimization is the process of modifying a software system to make some aspect of it work more efficiently or to use fewer resources. [Sed84, p. 84]

Optimization can be done on many different layers such as manually on the source code or automatically during compilation time by an optimizing compiler, by applying different manipulations, that are known to improve the performance of the program. [Lat]

Another form of compiler based optimizations is profile guided optimization (PGO), which uses additional information, gained from profiling the application, to optimize the generated code even further.

The following optimization techniques are subject to automation in this thesis:

2.3.1 Function Inlining

Function inlining reduces the overhead associated with calling and returning from a function by expanding its body in place of the function call and is often used to expose additional opportunities for optimization. [com]

Most optimizing compilers can inline functions to speed up the generated code but when done manually, it decreases the readability and maintainability by adding duplicate code.

2.3.2 Loop Fusion

Loop fusion combines two adjacent loops without data dependencies to reduce the loop overhead and enable the compiler to optimize the statements in both loop bodies together. Although loop fusion reduces loop overhead, it does not always improve runtime performance and may even reduce it by forcing the compiler to make bad use of the memory architecture. [com]

Most optimizing compilers are able to perform some kind of loop fusion, but not all, as some require extensive analysis of the dependencies between the loops or are not able to determine if fusing the loops would improve or reduce performance.

2.3.3 Loop Unswitching

Loop unswitching transforms a loop containing a loop-invariant if(-else) statement into an if-else statement, that contains different versions of the loops with the respective body of the old if(-else) statement in its body. [com]

Loop unswitching is a powerful optimization technique, that improves the performance by removing unnecessary computation and branching but also reduces the readability and maintainability as it creates different copies of the loop body.

3 Related Work

This chapter presents OpenMP, the Groovy compiler, GCC, LLVM and ROSE as examples for different tools to increase code quality and/or performance.

3.1 OpenMP

OpenMP (Open Multi-Processing) is an API that supports multi-platform parallel programming in C/C++ and Fortran. It is used to parallelize programs by distributing sequential blocks of code (mostly loops) to multiple threads, based on special compiler directives given by the developer. [ope]

The use of OpenMP can significantly ease the development process and increase the code quality, as it allows the developer to write a correct sequential program first and then increase its efficiency by incrementally parallelizing it without changing much of its code.

3.2 Groovy

The Groovy compiler allows the AST to be changed during compilation time. Special annotations such as '@Immutable' can be used to activate the AST transformations, which for example add all the necessary functionality to a class to set it *immutable* and reduce the amount of architectural code, that is not directly part of the functionality, similar to aspect oriented programming. [gro]

The elimination of otherwise necessary boilerplate code, reduces the size of the program and speeds up development.

3.3 GCC

The GNU Compiler Collection (GCC) contains frontends for high level languages like C, C++ and Fortran and can target a variety of different architectures. GCC can be set to automatically optimize the program during the compilation via the use of the '-O' option. [gcc]

Using an optimizing compiler like GCC can significantly increase the efficiency of a program, but does not fully replace the need for manual optimization as the compiler is limited by its implementation in terms of optimization options.

3.4 LLVM

LLVM is a compiler framework like GCC and offers similar optimization options and capabilities. One of LLVMs frontends translates the program to the LLVM IR (immediate representation), which then gets optimized and used to generate machine code. [Lat]

3.5 ROSE

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C, C++, UPC, Fortran, OpenMP, Java, Python and PHP applications. ROSE uses an AST based immediate representation for its translation process. [ros]

4 Design

This chapter describes the design process of the prototype. It explains the functionality and architecture of the prototype. Additionally it describes the scrapped first approach and the possible uses of the suggested prototype.

As briefly mentioned in the introduction, the prototype will be used to examine the feasibility of an AST-based approach of high level code manipulations. The prototype itself is supposed to aid development by automatically performing otherwise manually performed code optimizations at the source code level based on hints given by the developer.

4.1 Requirements

The requirements for the prototype are oriented towards what would be needed from the translation tool in a production environment to ensure the evaluation is not narrowed down to the problem specific requirements of the prototype itself.

For this purpose the requirements of the prototype are the following:

- **Functionality**. While the resulting optimized code is required to be as efficient as the manually optimized code itself, the unoptimized version has to be more read- and maintainable.
- **Integratability**. To ease the use and adoption of the prototype, it should be integrable in existing general purpose build systems (like GNU Make). This requires controllability via the command line and does not require any other user input except the program source code.
- **Extensibility**. The prototype should be light-weight as well as easily maintain- and extensible, so that the users (e.g. the scientists) can add support for additional programming languages and types of optimizations themselves.

4.2 Functionality

The prototype reads the file containing with DSL statements enriched source code, transforms it and writes it to a file.

The transformation is done in the following steps:

1. Parsing (Code –> AST). The input to the tool is program source code with DSL statements, naming which optimizations shall be performed, which it parses to an AST.
2. AST-Manipulation (AST –> AST). Then it optimizes the code by transforming the AST, if the individual optimizations are viable.
3. Unparsing (AST –> Code). At last it unparses the AST and outputs the resulting code.

4.3 Architecture

The architecture of the prototype went through one major revision, before getting realizable.

4.3.1 Initial Approach

The first idea was based on the architecture of today's compiler frameworks and planned to make the AST and thus also its manipulations language independent. This architecture would have therefore only required to write a new parser and unparser to add support for a new language. To make the AST language independent, its nodes would have to match the most common feature sets of (at least all supported) modern procedural programming languages, like functions and function calls, conditional and repetitive execution, assignments and variable access as well as expressions.

Soon this idea was scrapped, as the effort required to implement a parser and unparser would have outmatched the scope of this thesis:

- Parsers would not only have to parse its corresponding language source code but also may replace some syntax structures, which are not directly available in the AST, with others.
- Unparsers would not only be required to correctly unparse the AST to code, but should also be able to recreate their langague special elements and structures where applicable.
- Merely the combination of these parsing and unparsing capabilities would be able to translate every program freely between the supported languages.

4.3.2 Revised Approach

The second design approach dropped the idea of translating and manipulating the code in the form of a AST for pseudo code. It was replaced with the less complicated option of having a specific ASTs and manipulations for each language.

Which then lead to the prototype being split into three different components:

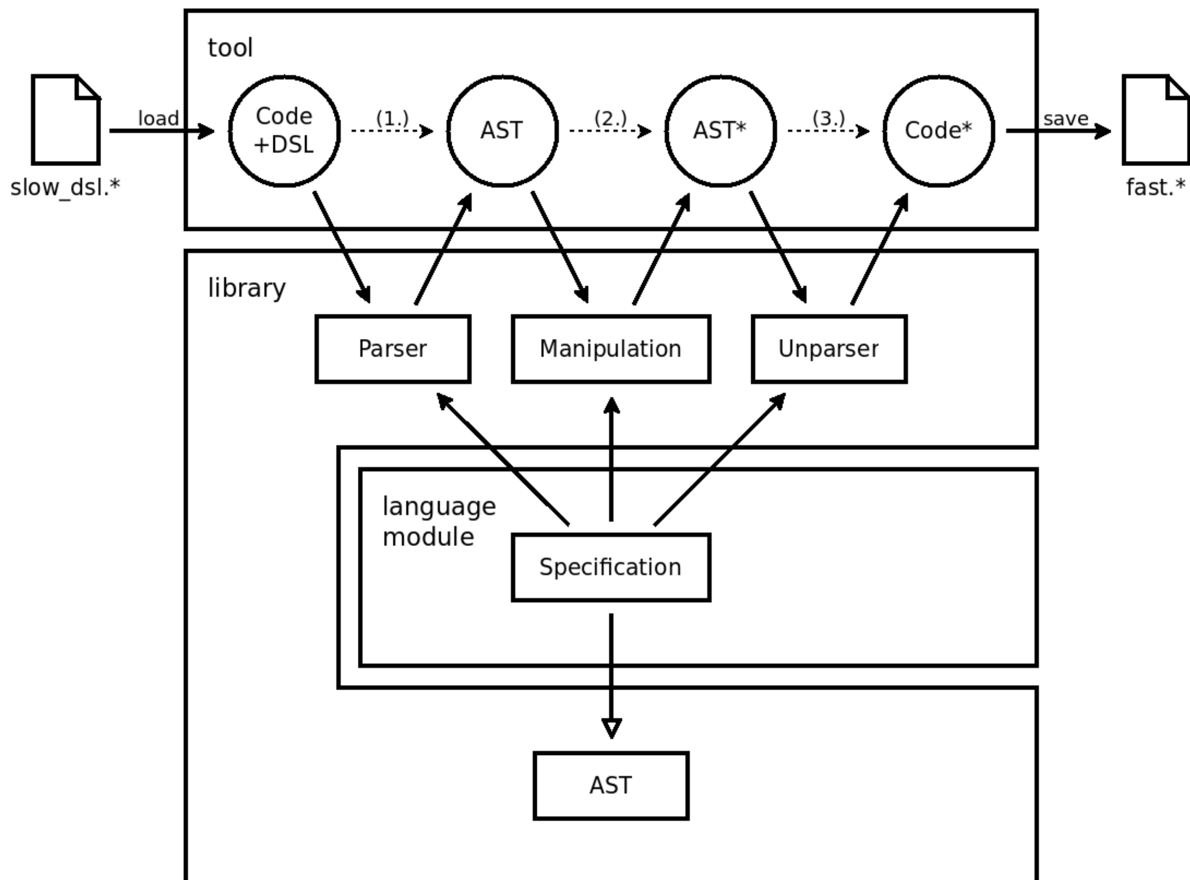

Figure 4.1: revised design approach

- A **library**, which contains an generic implementation of an AST and offers a language independent interface for parsing, unparsing and executing high level AST manipulations.
- A **plugin** for the library, which implements one language specific parser, unparser as well as a collection of AST manipulations.
- A **tool**, that wraps the capabilities of the library and its plugin into a simple command line tool.

4.4 Possible Uses

The library could be used in more complex tools, which for example could be used to automate the optimization process even further. For example these tools could follow either a simple brute force approach, which tests every possible optimization at a given time, or the more sophisticated way of using intelligent heuristics or profile-guided optimization.

5 Implementation

This chapter describes the implementation of the prototype. It illustrates the methodology and architecture of the implementation as well as the implementation of the abstract syntax tree, the parsing and unparsing processes. Additionally it describes the library plugin used in the evaluation.

5.1 Methodology

The prototype has been developed in several stages, based on the dependencies between the individual components. The first step was to implement the generic AST, since it is used by every other component. The first set of language specific components were implemented for a fictional language, named SimLang, to explore the difficulties of parsing a proper programming language and finding a suitable implementation of the language module. The syntax of SimLang was an appropriately chosen mixture of C and Python and once the parser was working, the unparser got implemented as well. Since the syntax was the only part of SimLang, which was properly specified, the implementation of the AST manipulations for SimLang was pointless as they could not be evaluated. Therefore, based on the experience gained from implementing the parser and unparser for SimLang, the language module for a subset of C99 got implemented in the same order. The parser was the first part to be implemented, the second part was the unparser based on the AST generated by the parser and when parsing and unparsing were working, the AST manipulations and finally the DSL processing got implemented.

The language of choice for the development of the prototype was Python (version 3.4.3), as its high-level built in data structures, dynamic semantics, interpreted and object-oriented nature make it very attractive for rapid development of prototypes.

5.2 Architecture

Influenced by Pythons freedom of choice regarding the used programming paradigms, the implementation is mostly written procedural, but uses object orientation where appropriate.

The prototype consists out of two fixed, one loose and one external module, that is not part of the standard library:

- The `pydslcc` module chains together the individual working steps (reading the source file and parsing the code, calling the execution of the language specific AST

manipulations, unparsing the AST and saving the result in a new file).

- The `util` module contains the object-oriented implementation of a generic abstract syntax tree.
- Each language module (for example `c99`) extends the implementation of the generic AST for its language, it also contains the specification for its languages parser as well as the manipulations and the code, which processes the DSL statements. It gets dynamically imported by the `pydslcc` module, based on an argument in the program call.
- The external `PLY` module is used by the `pydslcc` module to generate the parser for a language modules programming language (dialect).

5.3 Abstract Syntax Tree

The generic AST is implemented as a double linked tree with instances of the `Node` class as it nodes. Each `Node` object in the AST holds a reference to its parent node (except for the root node) and a list of references to each of its children (except if the node is a leaf). In addition to the references to its neighbours, each `Node` object also contains the value it represents (e.g. the identifier `i`) and its node type as a string (e.g. `IfStatement`). The `Node` class defines several methods like `append_child()` and `insert_child_rel()`, that are used during the parsing and manipulation process to create the references between the nodes. It also offers some Python internal convenience methods that ease the programmatic handling of individual nodes, for example the implementation of the `__getitem__()` method shortens the code for accessing a specific child of a node.

The `Node` class is nested in the `AbstractSyntaxTree` class, which is a higher level interface to access the tree as a whole instead per node. It holds a reference to the root of the tree of `Node` objects and provides the generic methods to pretty print, unparse and get a list of all nodes in the tree.

The specific AST of each language module is based on this generic implementation: Each syntax structure is implemented as a subclass of the `Node` and implements the `parse()` and `unparse()` method, which are respectively used for parsing and unparsing.

The specific AST of each language module is based on this generic implementation: Each syntax structure is implemented as a subclass of the `Node` and implements the `parse()` and `unparse()` method, which are respectively used for parsing and unparsing.

5.3.1 Types of AST Nodes

The implementation of Simlang and the C99 subset lead to the classification of the language specific AST nodes in four abstract categories:

- **Block node**. A block represents a sequence of syntactical independent structures, which in readable source code are vertically distributed on individual lines. In C

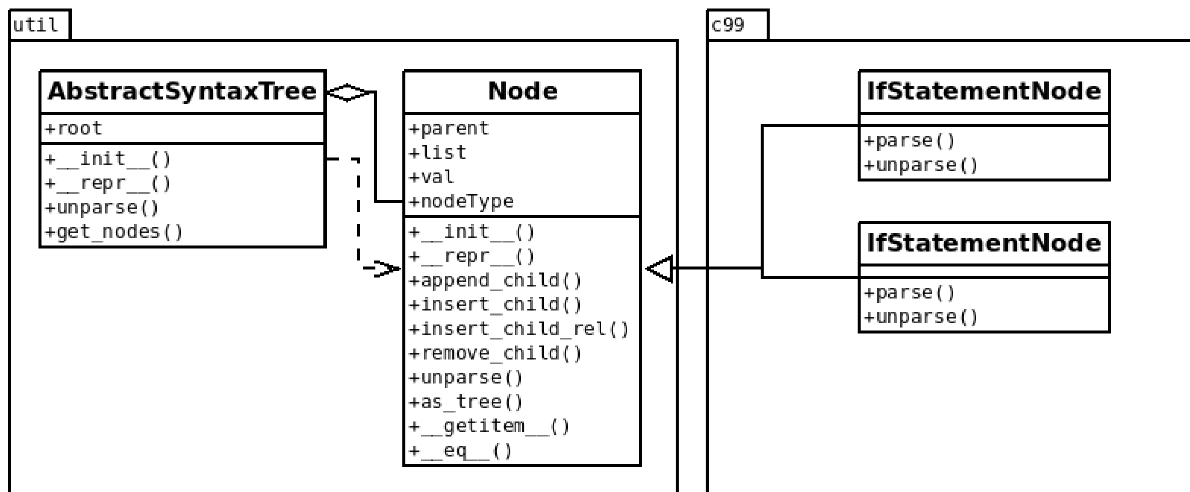

Figure 5.1: AST implementation

this category includes for example *statement blocks*, which are are also surrounded by curly brackets.

- **List node**. A list represents a list of syntactical independent structure, which in readable source code are written on the same line. In C this category includes for example *argument lists*, which are surrounded by parentheses and seperated by commas.
- **Value node**. A value node holds a single syntax element, which can not be divided any further. In C this category includes for example *constants*, *strings* or *identifiers*, but not brackets or operators as these are no values in itself.
- **Structure node**. Each structure is a syntactical unique sequence of elements, which can not be categorized otherwise. In C this category includes for example *if statements* or *expressions*.

5.4 Parsing

Each language specific parser is not statically implemented but dynamically generated during execution by the use of PLY, developed by David Beazley, based on the specification in the language module.

PLY (Python Lex-Yacc) is a pure-Python implementation of the compiler construction tools lex and yacc and consists of two separate modules. The `ply.lex` module is used to break input text into a collection of tokens specified by a collection of regular expression rules. The `ply.yacc` module is used to recognize language syntax that has been specified in the form of a context free grammar. PLY uses reflection (introspection) to build its lexers and parsers, which use the LALR(1) algorithm. [Bea]

The context free grammar as well as the instructions to build the AST are distributed among the parse methods of the `Node` subclasses. When the generated parser reduces a

term, it executes the respective method, which analyzes the term and creates and/or modifies the necessary objects. The argument of each `parse()` method is a list, which contains the references to the reduced objects, while the first element is initialy empty and used to save the reference to the object, which is the result of the reduce step.

```
class StatementBlockNode(util.AbstractSyntaxTree.Node):
    def parse(p):
        '''StatementBlock : Statement
                          | StatementBlock Statement'''
        if len(p) == 2:
            p[0] = StatementBlockNode()
            p[0].append_child(p[1])
        else:
            p[0] = p[1]
            p[0].append_child(p[2])

    # unparse method
```

Listing 5.1: C99 subset StatementBlockNode.parse() method

Listing 5.1 shows how a `StatementBlock` is either reduced from

- a single `Statement`, in which case the length of `p` is two and therefore a new `StatementBlockNode` object with the reduced `StatementNode` object as its child is the result.
- an existing `StatementBlock` and a `Statement`, in which case the length of `p` is three and the existing `StatementBlockNode` object is reused and the `StatementNode` object becomes its next child.

5.5 Unparsing

Each language specific AST unparses itself, which requires every `Node` subclass to provide a custom `unparse()` method.

The unparsing process follows a bottom-up approach: Node A calls the `unparse()` methods of its children and uses the returned syntactical equivalent source code to build itself. The children recursively do the same to convert themselves into source code.

```
class StatementBlockNode(util.AbstractSyntaxTree.Node):
    # parse method

    def unparse(self, prefix=''):
        code = ''
        for child in self.childs:
            code += prefix + child.unparse(prefix) + '\n'
        return code
```

Listing 5.2: C99 subset StatementBlockNode.unparse() method

Listing 5.2shows how a `StatementBlockNode` unparses itself, it creates a new string, appends each of its unparsed children, while taking care of the indentation and newline before and after each statement, and returns the string to the caller.

The `AbstractSyntaxTree` class provides an `unparse()` method as well, it calls the `unparse()` method of the root node.

5.6 C99 Subset

This section describes the language module for the subset of C99, that is used in the evaluation.

5.6.1 Supported Language Features

The following list of language features and elements are included in the C99 subset:

- *function definitions*
- *function calls*
- *global and local variable definition*, e.g. `int i;`
- *assignments*, e.g. `i = 4;`
- *combined definition and initialization* in one statement, e.g. `int i = 4;`
- *simple types*, `void`, `char`, `int`, `long`, `float` and `double`
- *array* syntax, e.g. `array[i]`
- *expressions* with support for ...
  - arithmetical infix operators (`+`, `-`, `*`, `/`, `%`)
  - logical infix operators (`&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=`)
  - bitwise infix operators («, »)
  - increment and decrement postfix operators (`++`, `-`)
  - prefix operators (`*`, `&`, `+`, `-`, `sizeof`)
- *if(-else) statements*
- *for statements*
- *return statements*
- *comments*, though inline comments will get unparsed on the next line
- *preprocessor pragmas*

5.6.2 AST Manipulations

Each optimization is split into two methods, one for analyzing and one for manipulating the AST. For example `inlineable(f, fc)` determines if the function `f` can be inlined in place of the function call `fc`, while `inline(f, fc)` inlines the function `f` in place of the function call `fc`.

The implementation of each optimization is limited to the set of specific cases, which are used in the evaluation. Therefore neither the validation nor the execution of the individual optimizations is complete.

Function Inlining

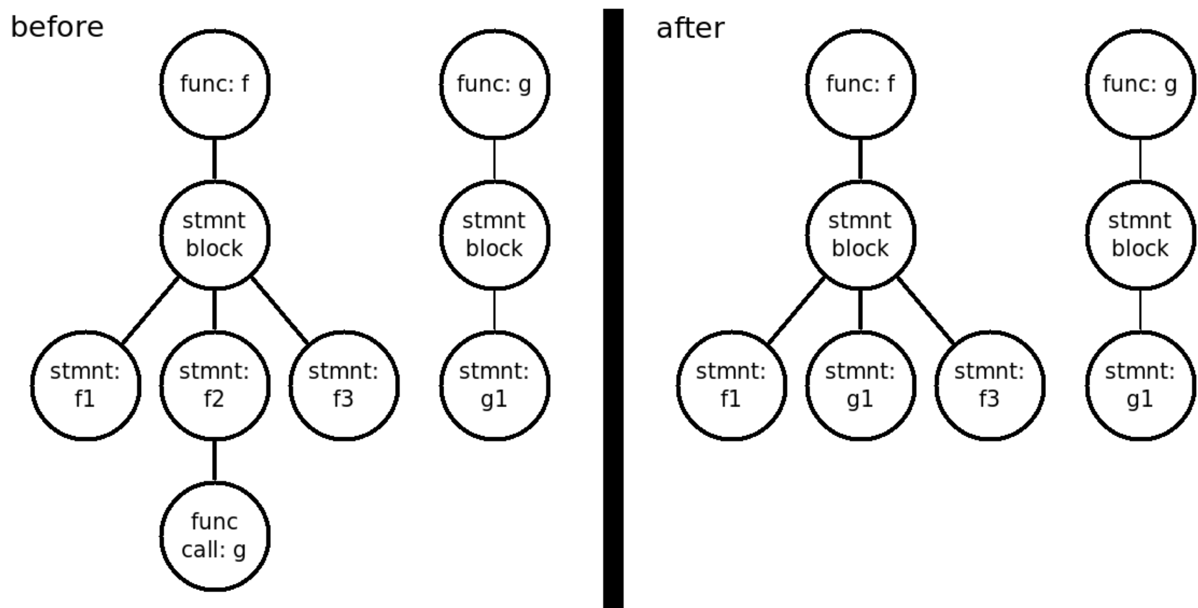

Figure 5.2: AST visualization of function inlining

The implementation requires that the function call in question calls the function to be inlined and that the function does not recursively call itself. Further it only supports inlining functions, which do not return a result, and ignores the arguments.

The manipulation consists of the following steps:

1. Copy the statement block of the function to inline.
2. Insert each statement in the copy before the call.
3. Remove the old function call.

Loop Fusion

The implementation requires that both for loops are on the same nesting level, that no statement is in between them and that the value ranges of the index variable match.

The manipulation consists of the following steps:

1. Rename the index variable of the second for loop if necessary.
2. Append the statements from the second loop to the first loop.
3. Remove the remains of the second loop.

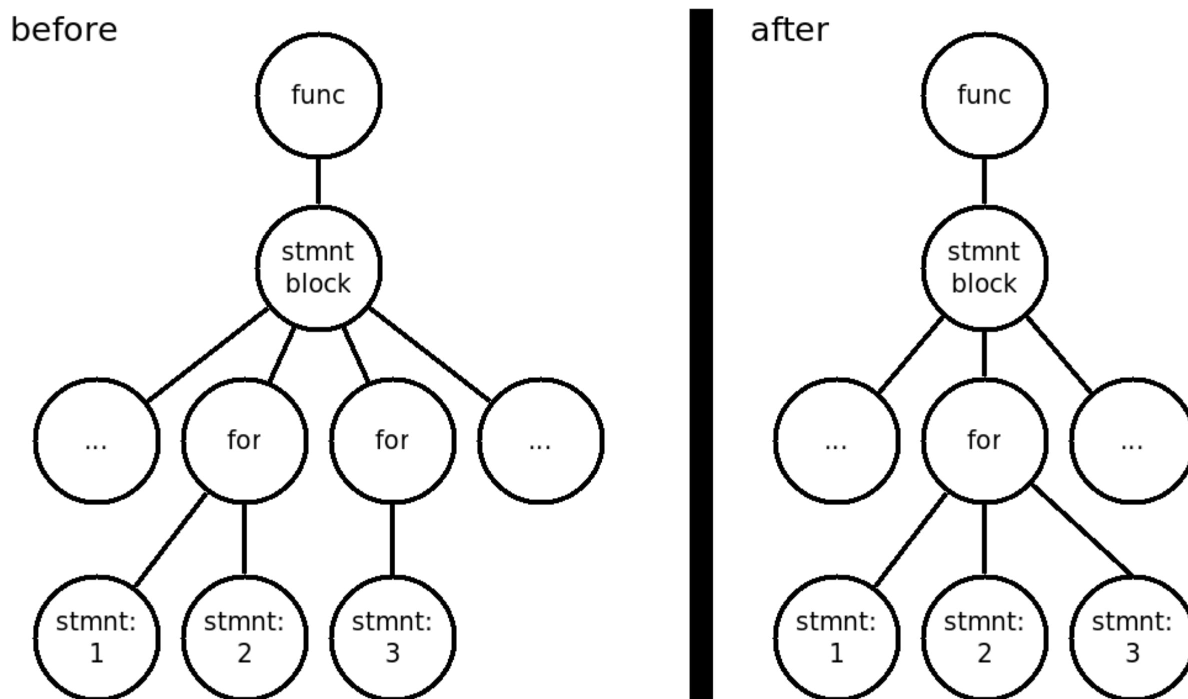

Figure 5.3: AST visualization of loop fusion

Loop Unswitching

The implementation checks if the if statement is directly in the statement block of the for loop.

The manipulation consists of the following steps:

1. Build the new if statement:
   a) Copy condition from old if statement.
   b) Build the *then* block, by copying the for loop and inserting the *then* block of the old if statement.
   c) Build the *else* block, by copying the for loop and inserting the *else* block of the old if statement, if it existed.

2. Insert the new if statement before the for loop.
3. Remove the old for loop.

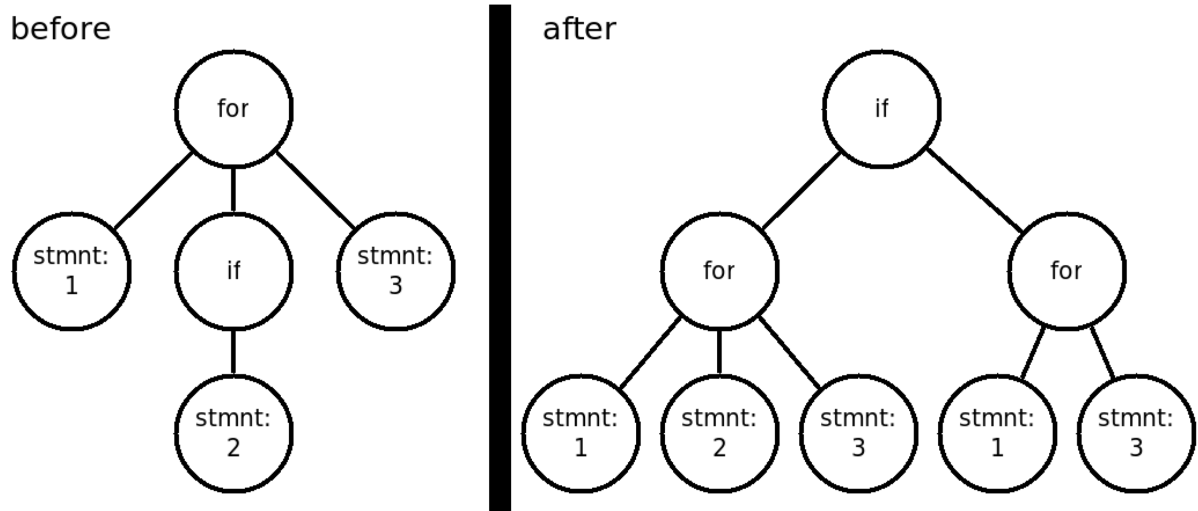

Figure 5.4: AST visualization of loop unswitching

5.6.3 DSL

The DSL defines the following statements:

- `#pragma dsl inline` instructs the prototype to check, if it can inline the next function call, that directly follows this instruction and do so if its possible
- `#pragma dsl fusion` instructs the prototype to check, if the next two loops, that directly follow this instruction, can be fused and do so if its possible
- `#pragma dsl unswitch` instructs the prototype to check, if it can unswitch the first if statement in the loop that directly follows this instruction and do so if its possible

To avoid a loss in readability of the source code due to possibly long chains of optimizations on individual lines, they can be also written on a single line, like `#pragma dsl inline fusion unswitch`, to reduce the required vertical space.

6 Evaluation

This chapter evaluates if the prototype meets the requirements and presents the insights in regards to the AST-based approach, the design of the prototype and implementation itself.

6.1 Requirements

6.1.1 Functionality

To be suitable for its purpose, the prototype has to optimize DSL enriched code, that is slow but read- and maintainable, so that it is at least as fast as the manually optimized code.

Four different versions of the same test program were used and evaluated comparing their readability, maintainability and performance to verify the functionality of the prototype:

- **slow**. The unoptimized reference implementation, which is focused on readability and maintainability.
- **slow_dsl**. The DSL enriched version of *slow*, which gets optimized by the prototype.
- **fast**. The manually optimized version of *slow*, which is focused on performance.
- **slow_dsl_optimized**. The optimized version of *slow_dsl*, which is produced by the prototype.

```
1  #define SIZE 400000000
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #include "mtime.h"
7
8  int array[SIZE];
9  int w;
10
11 void add_one() {
12     for (int i = 0; i < SIZE; i++) {
13         array[i] = array[i] + 1;
14     }
```

```c
15 }
16
17 void add_two_or_set_zero_when_w() {
18     for (int i = 0; i < SIZE; i++) {
19         array[i] = array[i] + 2;
20         if (w) {
21             array[i] = 0;
22         }
23     }
24 }
25
26 void test() {
27     w = 0;
28     add_one();
29     add_two_or_set_zero_when_w();
30     w = 1;
31     add_two_or_set_zero_when_w();
32 }
33
34 int main() {
35     repeat(test);
36 }
```

Listing 6.1: slow.c

```c
1  #define SIZE 400000000
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #include "mtime.h"
7
8  int array[SIZE];
9  int w;
10
11 void add_one() {
12     for (int i = 0; i < SIZE; i++) {
13         array[i] = array[i] + 1;
14     }
15 }
16
17 void add_two_or_set_zero_when_w() {
18     for (int i = 0; i < SIZE; i++) {
19         array[i] = array[i] + 2;
20         if (w) {
21             array[i] = 0;
22         }
23     }
24 }
25
26 void test() {
27     w = 0;
28     #pragma dsl inline fuse unswitch
```

```
29      add_one();
30      #pragma dsl inline
31      add_two_or_set_zero_when_w();
32      w = 1;
33      #pragma dsl inline unswitch
34      add_two_or_set_zero_when_w();
35  }
36
37  int main() {
38      repeat(test);
39  }
```

Listing 6.2: slow_dsl.c

```
1   #define SIZE 400000000
2
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   #include "mtime.h"
7
8   int array[SIZE];
9   int w;
10
11  void add_one() {
12      for (int i = 0; i < SIZE; i++) {
13          array[i] = array[i] + 1;
14      }
15  }
16
17  void add_two_or_set_zero_when_w() {
18      for (int i = 0; i < SIZE; i++) {
19          array[i] = array[i] + 2;
20          if (w) {
21              array[i] = 0;
22          }
23      }
24  }
25
26  void test() {
27      w = 0;
28      if (w) {
29          for (int i = 0; i < SIZE; i++) {
30              array[i] = array[i] + 1;
31              array[i] = array[i] + 2;
32              array[i] = 0;
33          }
34      } else {
35          for (int i = 0; i < SIZE; i++) {
36              array[i] = array[i] + 1;
37              array[i] = array[i] + 2;
38          }
39      }
```

```
40     w = 1;
41     if (w) {
42         for (int i = 0; i < SIZE; i++) {
43             array[i] = array[i] + 2;
44             array[i] = 0;
45         }
46     } else {
47         for (int i = 0; i < SIZE; i++) {
48             array[i] = array[i] + 2;
49         }
50     }
51 }
52
53 int main() {
54     repeat(test);
55 }
```

Listing 6.3: fast.c

```
1  #define SIZE 400000000
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "mtime.h"
5  int array[SIZE];
6  int w;
7  void add_one() {
8      for (int i = 0; i < SIZE; i++) {
9          array[i] = array[i] + 1;
10     }
11 }
12 void add_two_or_set_zero_when_w() {
13     for (int i = 0; i < SIZE; i++) {
14         array[i] = array[i] + 2;
15         if (w) {
16             array[i] = 0;
17         }
18     }
19 }
20 void test() {
21     w = 0;
22     if (w) {
23         for (int i = 0; i < SIZE; i++) {
24             array[i] = array[i] + 1;
25             array[i] = array[i] + 2;
26             array[i] = 0;
27         }
28     } else {
29         for (int i = 0; i < SIZE; i++) {
30             array[i] = array[i] + 1;
31             array[i] = array[i] + 2;
32         }
33     }
34     w = 1;
```

```
35      if (w) {
36          for (int i = 0; i < SIZE; i++) {
37              array[i] = array[i] + 2;
38              array[i] = 0;
39          }
40      } else {
41          for (int i = 0; i < SIZE; i++) {
42              array[i] = array[i] + 2;
43          }
44      }
45  }
46  int main() {
47      repeat(test);
48  }
```

Listing 6.4: slow_dsl_optimized.c

The DSL statements in *slow_dsl* marginally reduce the readability and maintainability of the code compared to *slow* about as much as OpenMP pragmas, to direct the parallelization, do. Yet the reduction is to a significantly lesser extent than the changes from the manual optimizations in *fast*, which lead to duplicated and less structured code. Since *slow_dsl_optimized* is only supposed to be generated as part of the compilation, its structural code quality is of no relevance for the developer.

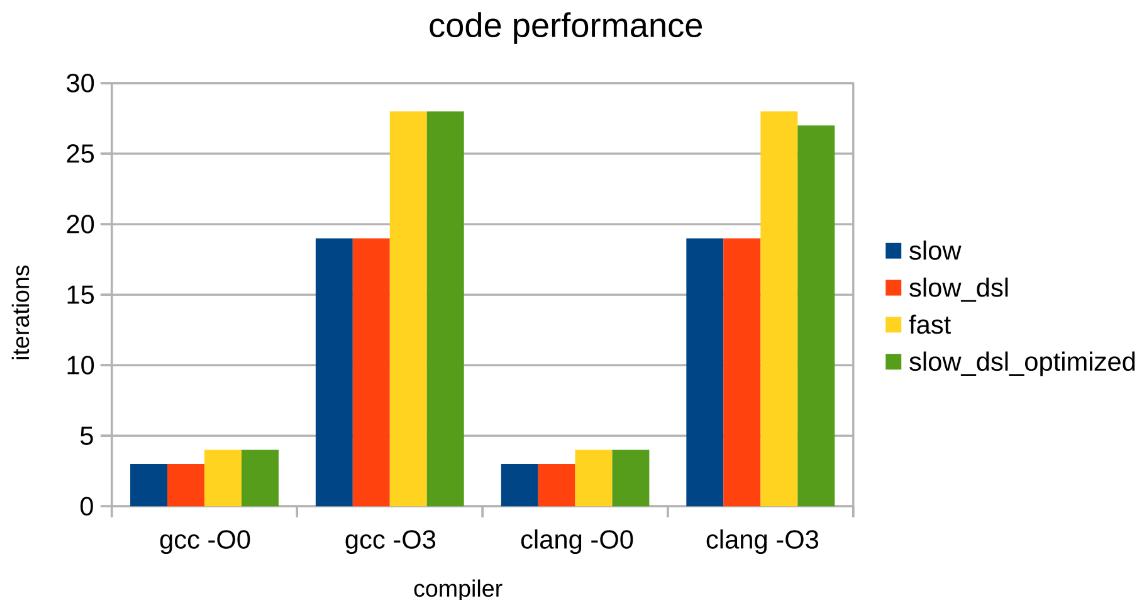

Figure 6.1: code performance measurements

The comparison of the performance was done by measuring the number of executions of the `test()` function for a duration of ten seconds. The programs were compiled with four different combinations of compilers (GCC and CLANG) and optimization options

(-O0 and -O3). The measurements were taken on a Lenovo ThinkPad X230i with a Intel i3-3110M CPU and performed at least three times to compromise for differences in the workload of the system.

The measured performance, shown in Figure 6.1, is the same for both unoptimized and manually optimized versions, regardless of the chosen compiler and optimization options. The performance difference between *fast* and *slow_dsl_optimized*, using CLANG with -O3 enabled, arises from averaging and rounding the measured numbers and does not imply that *fast* is faster than *slow_dsl_optimized*.

6.1.2 Extensibility

Based on its design and the dynamic nature of Python, extending the prototype in terms of supported languages and optimizations is quite simple:

- Adding a new language requires the implementation of the corresponding new language module. The language module is programmatically imported, by the use of the `importlib` Python module and the modules name, specified as the first argument on the command line.
- Adding new optimizations requires the implementation of the corresponding AST manipulations and adapting the parser of the used DSL.

Since each language modules is a fully functional Python modules, it does not burden the developer with learning an additional special language, which is exclusively used for only this purpose, before writing the extension in itself.

6.1.3 Integrability

The call required to optimize *slow_dsl* was `pydslcc.py c99 slow_dsl.c` and did not need any other input or arguments than the file `slow_dsl.c`. In principle the DSL-instructed approach is utilizable in larger projects, but would most likely require changes to the project structure and configuration of the build system, since the performed optimization step is not done by the compiler and therefore switches the file the compiler has to translate to build the application. Another possible way is to pipe the resulting code directly to the compiler, which omits any intermediate file.

6.2 Design & Implementation

6.2.1 Performance

The execution time of the prototype should be as small as possible, as it has to be run for each source file, which has changed. The longer the execution takes the slower is the build process which ultimately would slow down the development.

The performance of the prototype was measured for different code lengths of *slow_dsl*, by appending it multiple times to itself. To measure the duration of the individual working steps the prototype was executed with and without DSL-directed manipulations enabled.

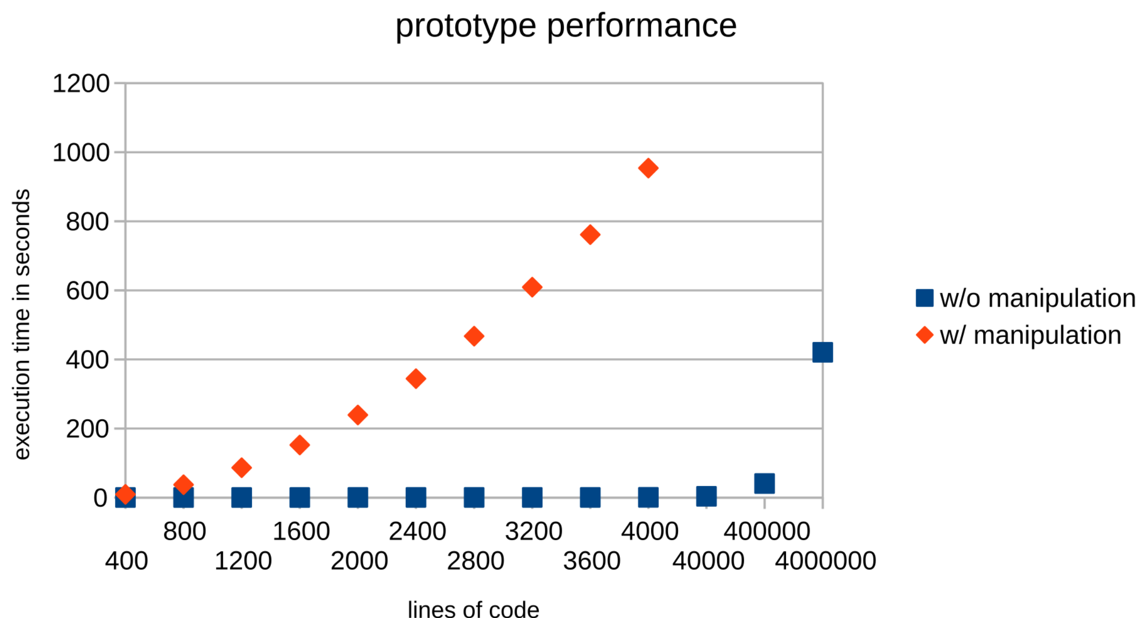

Figure 6.2: performance of the prototype

As seen in Figure 6.2, the time required for parsing and unparsing increases linear with the length of the code or respectively the AST, as does the theoretical complexity of both operations. The time required for parsing, unparsing and performing the manipulations increases polynominal with the length of code, although it should not, as the effort for each manipulation is bounded by the size of the AST subtree it modifies. This difference, caused by the execution of the AST manipulations, suggests that their implementations perform unnecessary work and are therefore not as efficient as possible.

This implementation specific performance problem would have to be found and resolved for the prototype to be usable with longer source code files.

6.2.2 AST-based Approach

Using an AST as the representation for the source code works well for syntactical analysis and manipulations, as it explicitly outlines the structure of the code and is easier to work with than a textual representation. However ASTs do not natively support all types of code analysis and are especially inappropriate for the ones, that are based on the execution of the code instead of its syntactical semantic, such as the control flow and data dependencies between individual statements.

To be able to thoroughly test the validity of the optimization, suggested by the developer, more specialized models of the code are needed, which on the other hand would increase the complexity and size of the tool.

7 Conclusion

7.1 Summary

This thesis investigated the possibilities and obstacles of automating manual code optimizations by performing them as DSL-directed AST transformation. A Python based prototype was used to show the feasibility of the AST-based approach and that DSL-directed code transformations can be used as an additional step in the compilation process to decouple code quality and performance and therefore reduce the effort required to continuously develop optimized applications. The implementation of the prototype further revealed, that modelling the optimizations as AST transformations for a real world high-level programming language becomes increasingly more complex proportional to the number of possible edge cases and types of dependencies, that have to be checked.

7.2 Future Work

The future steps in developing a comprehensive prototype would be to improve the implementation of the AST manipulations by finding and extracting the generic building blocks of the manipulations as well as improving the API of the AST and Node implementation. In addition support for optimizations as well as the edge cases of the currently implementend ones and more sophisticated dependency analysis should be added to the prototype. A later step would be a profound test of the practicability of this approach by using it in the development process of an exemplary real world application.

Bibliography

[aim] AIMES. https://wr.informatik.uni-hamburg.de/research/projects/aimes/start. Accessed: 2016-06-26.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullmann. *Compilers : principles, techniques, & tools*. Pearson, Addison Wesley, Boston u.a., 2. ed., pearson internat. ed. edition, 2007.

[Bea] David Beazley. PLY (Python Lex-Yacc). http://www.dabeaz.com/ply/. Accessed: 2016-06-26.

[com] Compiler Optimizations. http://www.compileroptimizations.com/. Accessed: 2016-06-26.

[gcc] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. Accessed: 2016-04-30.

[gro] Immutable. http://docs.groovy-lang.org/next/html/gapi/groovy/transform/Immutable.ht Accessed: 2016-04-29.

[Lat] Chris Lattner. The Architecture of Open Source Applications: LLVM. http://www.aosabook.org/en/llvm.html. Accessed: 2016-06-26.

[ope] OpenMP.org. http://openmp.org/. Accessed: 2016-04-30.

[ros] ROSE compiler infrastructure. http://rosecompiler.org/. Accessed: 2016-06-26.

[Sed84] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1984.

Appendices

List of Figures

1.1 AIMES' approach to enhance programmability 7

4.1 revised design approach 15

5.1 AST implementation 18
5.2 AST visualization of function inlining 21
5.3 AST visualization of loop fusion 22
5.4 AST visualization of loop unswitching 23

6.1 code performance measurements 28
6.2 performance of the prototype 30

List of Listings

5.1 C99 subset StatementBlockNode.parse() method 19
5.2 C99 subset StatementBlockNode.unparse() method 19

6.1 slow.c 24
6.2 slow_dsl.c 25
6.3 fast.c 26
6.4 slow_dsl_optimized.c 27

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

____________________ ______________________________

Ort, Datum Unterschrift