Report

LLVM-Infrastructure

written by

Jan Moritz Witt

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

Studiengang:	Meteorologie
Matrikelnummer:	7174687
Betreuer:	Michael Kuhn

Hamburg, 2020-01-23

Contents

1 Introduction

This report is based on the presentation about the LLVM infrastructure held by me in the seminar 'Effiziente Programmierung' in November 2019.
LLVM is an open source compiler project with a modular library-based structure, which offers the opportunity to use only the wanted features and optimisations while compiling. While written in C++ it can be used for several programming languages and in the last decade its tasks and offers expanded significantly, which will be presented in this report. To understand its purpose this report will start with a general introduction into compilers, going over different parts on an example and explaining an intermediate representation, while mostly focusing on the used structure from the LLVM compiler. Afterwards there is a summary of LLVM and its core projects, ending with the larger sub-projects.

2 What is a compiler?

A compiler in general is a translator. It translates a source code mostly written in a higher-level programming language and understandable for human programmers into binary code machine language so the processor can understand and execute it. While translating the code the compiler also checks for error based on the rules of the used programming language and performs optimisations to them. If no errors occurred the compiler generates an object file for this one specific processor. This file is self-containing and can be executed without further application or package and as often as wanted any calculation time. Languages that are compiler based are for example C and C++.
Another way used by some programming languages to execute a source code is with an interpreter. An interpreter directly executes the source code line by line without generating an object file. Therefore, the entire code has to be translated every time again. Example languages are Python and Matlab.

While probably the most used compiler translates from a higher-level programming language to binary code to execute the written code on the used computer, it is not the only kind of compiler.
The cross-compiler is generating the object file on a other type of processor than the object file can be executed on. The decompiler goes the other direction than the normal compiler and generates a higher-level programming language source code from an object file. The source to source compiler translates a source code into a source code in another programming language.

The compiler structure can be divided into one or multi pass compilers. They describe the amount of models defined to do the translation and the included steps. The one pass does everything in a single module. Starting by analysing the source code and looking for errors, and ending with optimizations and translation into the processor specific binary language. The first compilers were one pass compilers, because they are smaller than the multi pass compilers and fitted better the limitations of early computers. They were also faster, due to the fact that they calculated only once, which also limits the amount of optimizations possible.
The multi pass compiler processes the source code multiple times, which helps to optimise the code. Therefore this is the structure almost all compilers nowadays possess. It can be divided into two main passes, the Front end and the Back end [5], which will be separately explained in more detail. In general, Front end are written for different source code languages, going together to a source and target independent form, the

intermediate representation, while the Back end are translations from this independent form to different target languages. This has the benefit, that the independent form can be shared between many different source and target languages. Sometimes, the independent part is referred as Middle end.

2.1 Front end

The Front end takes the source code as input and translates it into an intermediate representation of it. While translating it checks for errors and and writes down warnings if existing.
The Front end can be described as three different analysis, the lexical, the syntax and the semantic analysis. The Front end will be exemplarily explained with a simple C code:

```c
int mul_add(int x, int y, int z){
    // This example function multiplies x and y and add z
    // afterwards and returns the solution
    return x * y + z;
}
```

Listing 2.1: Example code taken from [8] but slightly changed by adding a comment

The code describes the definition of a function *mul_add* with three integers x, y, and z as input. The output is an integer calculated by the multiplication of x with y and the addition with z. This example is taken from the LLVM official web page, slightly changed by adding a describing comment for the function.
If all phases of the analysis are completed the code has no further errors how it is written and the code is translated into the intermediate representation. This is similar for all compiler of a compiler project, so optimizations can be shared between all different source code languages.
In the following subsections the analysis phases are explained in more detail and shown on the example code in Listing 2.1.

Lexical Analysis

The lexical analysis is the first phase of verifying the correctness of the source code. It checks if the code consists of the wanted language [1]. The language has defined sequences of characters with defined meanings, often called lexemes. A sequence of characters is called token. The lexical analyser goes over the code and tries to transform it into a sequence of tokens, which is used from the syntax analyser later on. Our example code (Listing 2.1) would be transformed by a lexical analyser to

```
'int' 'mul_add' '(' 'int' 'x' ',' 'int' 'y' ',' 'int' 'z'
  ↪ ')' '{' 'return' 'x' '*' 'y' '+' 'z' ';' '}'
```

Listing 2.2: Sequence of token

Tokens can be distinguished in their role in the code. Common are keywords, identifier, operator, separator and literals like constants, booleans and strings. Whitespaces, tabs and comments have no further meaning and use in a programming language and are mostly to separate different tokens and to make the code readable and understandable for programmers. They are ignored by the lexical analysis. In our example the tokens are defined as:

- Keyword: $int, return$
- Identifier: mul_add, x, y, z
- Operator: $(,), \{, \}, *, +$
- Separator: , , ;

A character sequence, which cannot be defined as a valid token in the source language raises a lexical error. Examples are variable names which start with a number or include a special symbol other than the underscore.
If the sequence of tokens over the code can be generated the lexical analysis was successful and completed and the syntax analysis begins.

Syntax Analysis

The syntax analyser takes the sequence of tokens from the lexical analysis as input and introduces the grammar of the specific language to the tokens. The syntax analysis is also often called parsing [2]. The syntax analyser or parser is verifying the structure of the source code. Errors that are found here are imbalances of opening and closing brackets and in case of for example C if every variable has an existing defined type. In case of our example code a syntax error would occur if the result of the function wouldn't have been defined as an integer in row 1. If syntax/grammar of the code is correct, the syntax analyser/parser is developing a syntax/parse tree based on the predefined grammar of the language. For our example this would look like

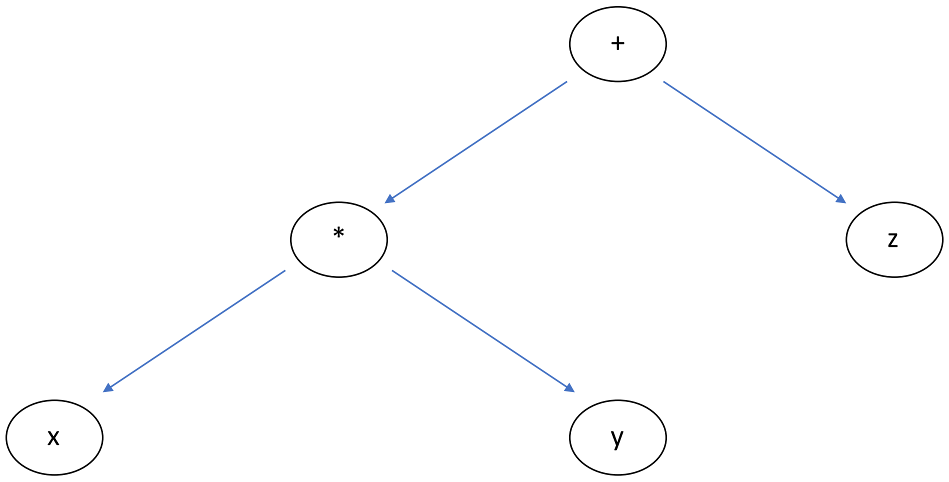


Figure 2.1: Parse tree of code example return command

The highest point in graphic representation is called root, points further down leaves. There exist two different types of parsing techniques, top-down and bottom-up parsing, which stands for direction the parser is heading to develop the parse tree.
When the parse tree can be developed without errors the grammar of the language is fulfilled and the syntax analysis is completed. The parse tree is used for the semantic analysis.

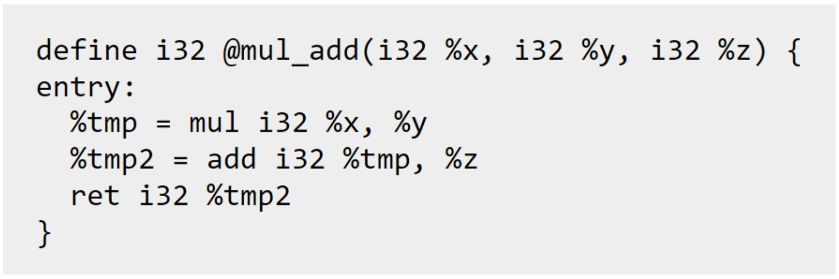
Semantic Analysis

The semantic analysis is the last of the three phases of the Front end. It checks if the code is semantically correct by using the sequence of tokens of the lexical and the syntax tree of the syntax analysis [3].
So far the code has been verified that the code consists of known sequences of characters and they are combined in a way corresponding to definition in the used language. Looking at our example in Listing 2.1, when our defined function *mul_add* have been called and variable x have been given the value of a *string* or a $float$ or haven't had a defined type at all, it wouldn't have raised an error yet. This is what the semantic analysis is for, it

checks the code for semantic correctness, e.g if the type matches. If this is the case the third phase is completed and the code is correctly written in the sense of the language definition and can be compiled. Of course, that does not mean the code is doing what the programmer wants it to do. These aren't errors but the definition of the language and have to be found by the programmer.

2.2 Intermediate Representation (IR)

When all three phases of the Front end have successfully finished the code has been verified and the code gets a first time translated. And here is probably the most important argument for compiler projects: All different Front end for specific source code languages translate into the same abstract machine language, the intermediate representation [9]. The intermediate representation is independent from the the source and the target language. This reusablility makes it much easier to expand the compiler project for more source and target languages and it simplifies the translation from source to target code by dividing the step into two.
The IR can be divided into two main structure types: hierarchical and flat IRs [7], [10]. A hierarchical IR uses nested structures, e.g. if-statements and do-loops, and therefore is closer to a high-level programming language. In a flat IR instructions are executed sequentially, which is closer to the structure of a processor. Also IR with a combination of these two forms are possible.
The intermediate representation of the LLVM project is a hierarchical IR with a SSA form. SSA stands for Static Single Assignment, which means every variable name can be assigned exactly once. This simplifies some optimisations and analysis but for the cost that the code expands heavily very easy. For our example code from the Front end (Listing 2.1) the code already expands to



```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

Figure 2.2: Example of the intermediate representation of LLVM for the C code example Listing 2.1 [8]

The intermediate representation is passed to the Middle End.

2.3 Middle end

The Middle end is defined as the part, where the compiler performs optimisations on the intermediate representation form of the source code, which are independent of the source and the target language [4]. The Middle end is the reused part of the compiler project. As a remark, in some literature the Middle end is not described as a separate part of the compiler but the target independent optimisations as part of the Front end before the optimised IR is passed to the Back end. An example for a compiler project and the shared optimisations in the Middle end is shown in figure 2.3.
Modern optimisations are structured as series of passes who transform the intermediate representation. The primary goal of the optimisations is to reduce the run time of the code without changing its meaning. Examples for machine-independent optimisations are

- Remove redundant calculations
- Discover and propagate constant values
- Discover and remove unreachable code
- Simplify control structure

After successfully transforming the IR for used optimisations the Middle end with the target independent optimisations are completed and the updated IR is passed to the Back end.

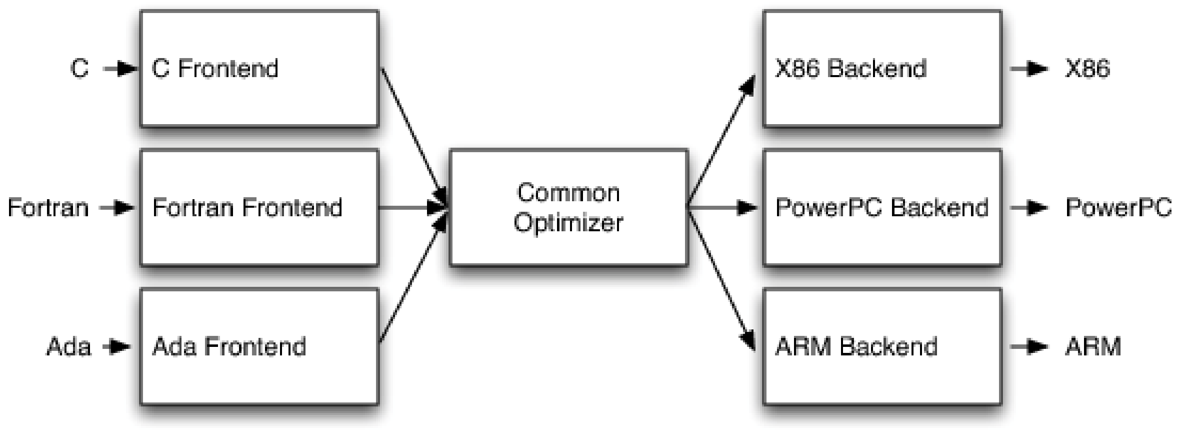


Figure 2.3: Exemplary of a compiler project. Different Source code languages use the same Middle end and IR, symbolised by the common optimizer, and then translate into different target languages. Taken from [18].

2.4 Back end

The Back end is the last part of this compiler structure. It takes the optimized IR from the Middle end and performs more, machine dependent optimisations before translating the final IR into the binary language code for the specific processor as an object file. Examples for further optimisations are

- replace complex operations with simpler ones
- register allocation in process
- instruction scheduling to increase parallel execution
- exploit memory hierarchy

This structure is one of the most common structures for compilers nowadays due to the mentioned benefit of sharing the optimisation part. Two large examples of compiler projects who use this structure are the Gnu Compiler Project (GCC) and the LLVM, which is the reason for explaining the structure in such a detail.

To understand the reasons behind developing the LLVM project and how it it different to the compilers mostly used back then an introduction into the GNU project is necessary. The GNU project was launched 1984 by Richard Stallman [11]. His reason was to build an UNIX-like open source software system, one software was the C/C++ compiler GNU C Compiler (GCC), which later supported also other languages and the name changed to GNU Compiler Collection. GCC is an official compiler for GNU and Linux systems. The software is released under the GNU General Public License (GPL) and GNU Lesser General Public License (LPGL). GCC performs well as a standard compiler, but it is not practical for specific requirements because the code is strongly coupled and the copyleft license is another limiting factor. [12]
Targeting these limitations was the idea behind developing LLVM. Today LLVM is approximately as common as GCC and many users see the advantage. The most prominent user is probably Apple, who switched from using GCC to the compiler Clang from LLVM.

3 LLVM

Figure 3.1: The logo of the LLVM umbrella project, taken from [8].

LLVM started as a research project by Chris Lattner and Vikram Adve at the University of Illinois, which was published in 2004 with the title "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", while the open source version was released 2003 [8]. Their goal was to develop a modern SSA-based compilation strategy for both static and dynamic compilations for any programming language. To reach that goal they built up LLVM as a collection of modular and reusable compiler and toolchain technologies.
LLVM originally stood for *Low Level Virtual Machine* but after its rapid growth after Apple Inc. hired them to build a compiler and technologies fitting to their needs for all their operating systems in 2005 and due to the open source status it got many different subprojects with a wide variety of purposes, so the name of the umbrella project was changed to just *LLVM*, without being synonymous to a longer name.
The code of the LLVM project is licensed under the "Apache 2.0 License with LLVM exceptions". This is the general Apache 2.0 License with two minor exceptions to ensure a very permissive licensing of LLVM. These exceptions allow to derive commercial products from LLVM without making them open source. While it appreciated to contribute changes on the LLVM back to the project it is not required. This means LLVM has no copyleft like GCC, where every copy, reproduction or adaptation of the code are under

the same license terms. This, together with a lack of necessary support of calling GCC as a module led companies like Apple, Intel, Sony or NVIDIA to use LLVM [12] and led to its success.
The next big milestone was in 2012, when LLVM has been awarded the ACM Software System Award by the Association for Computing Machinery [13]. In their reasoning they highlighted the variety of tasks, the clean and flexible design, the incorporations with companies and the use in science research. They pointed out that LLVM has replaced GCC as infrastructure of choice for doing research on program translation, optimization, and analysis. [14]
While LLVM started with C compiling, due to open source projects it can be used directly to compile languages like Ruby, Python, Java and PHP. And it can be done for simple JIT as well as for compiling Fortran code for supercomputers. Next to compiling code, it can be used for a variety of tasks such as optimisations, debugging, as a linker. There are a lot of official sub-projects, which will be introduced here in further detail, next to a list of ongoing open source projects, e.g. Julia, a new high-performance programming language for technical computing.
What makes it also very convenient to start with LLVM is the help it provides through documentations of the different parts and the design and tutorials for the start, next to its already clean structure. Last, the community of LLVM has to be mentioned. There are a lot of ways to get in contact for questions via a blog, a mailing list, discussions, meetings or even via a Discord channel. It is also possible to become developer and contribute to the projects yourself.

3.1 The main sub-projects

LLVM has a large community and lot of companies, scientific researchers and other programmers work with or on it. There are some projects LLVM is working on itself because, which will be presented here in more detail.

LLVM core

LLVM core is a set of libraries, which provide optimisations independent from the source and target language as well as the translation into the target language [8], i.e. the Middle and the Back end of a compiler. Libraries are a collection of pre-compiled routines a program can directly use. For the optimisations are built for the intermediate representation of the LLVM project, LLVM IR.
LLVM IR is used in all phases of the compilation. As mentioned before it has SSA form, which is helpful for many optimisations and analysis passes. It also provides type safety, low-level operations and flexibility and can be represent nearly all high-level languages. Also, LLVM IR is invented in three different but equivalent forms: as in-memory compiler IR, as on-disk bit-code representation for JIT compilers and a human readable assembly representation. The last one simplifies debugging and visualisation of the translation, while keeping an efficient compilation.

The optimisations of the source code in LLVM IR form are implemented as passes, i.e. one optimiser goes over the entire code and after this one is done the next starts by going over the entire, maybe already transformed, code for optimisations. The optimisations can be divided into three categories: Analysis passes, transform passes and utility passes [8].
Analysis passes compute information other passes can use for debugging or visualisation. Examples are

- -da: Dependence analysis
  Framework to detect dependencies in memory access
- -instcount:
  Counts all instructions and reports them

Transform passes change the code by using the information gathered by the analysis passes. Examples are

- -constmerge:
  Merge duplicate global constants into one shared constant
- -constprop:
  Replaces instructions for only constant operands into a constant value

Utility passes describe every pass, which give some sort if utility but doesn't fit into the first two categories. Examples are

- -verify:
  Verifies LLVM IR code. Helpful to run after optimisation testing
- -view-cfg
  Displays the control flow graph using the GraphViz tool.

After all optimisation passes are done the LLVM IR is probably changed in many ways the programmer wanted, is will be translated into the target language by using the code generator of the LLVM core.
The LLVM code generator is a target-independent framework that provides reusable components for translating the LLVM IR into the machine code of the specific target. It can be assembly form for static compilers or binary form for JIT compilers. For many targets the needed information for the code generator is already known and can be used, but otherwise it is explained in detail which information is necessary. The code generator can be divided in six main components, which all can be customised for the programmers demands, but which won't be explained here into further detail.
The LLVM core has again a large documentation about its structure and how to invent your own programming language and using the LLVM core as optimiser and code generator. But this is not topic of this report.

Clang

Clang is a Front end and tooling infrastructure specially for the C language family [8]. It has a library-based structure, which makes it straight-forward in time needed for compiling and has the big benefit that while it is compatible with GCC and its extensions, clang is able to load only the extensions wanted and ignore everything else. Further it has expressive diagnostics, i.e. warnings and errors pinpoint the problematic code and also highlights related informations.
One specific example for fast runtime is given by LLVM by comparing to GCC: "[Clang is]About 3x faster than GCC when compiling Objective-C code in a debug configuration" [8].
Clang can also be used for IDE, integrated development environment, which is helpful for software development. It has to be mentioned that the code base is hackable and a programmer can customize Clang. With the support of the comments it is simple to understand with a little knowledge of compilers.
Furthermore, Clang contains some tools to make coding and compiling easier. One tool is the Clang static analyzer. It is an source code analysis tool for the C family languages. It consist of algorithms and techniques to find code errors automatically. What makes it special is, that is also finds errors with normally are found by run-time debugging techniques. It has to be mentioned that is not free of errors and it can't be used for error checking alone. Also the static analysis can be much slower than compilation.

LLDB

LLDB is a high-performance debugger. LLDB converts debug information into Clang type, so they can directly be used by Clang. It also has a up-to-date language support for the C family languages, and is built as reusable components with support libraries of larger LLVM projects like the Clang expression parser. It is the standard debugger in Xcode in macOS.
Comparing itself again to the corresponding product of GCC, LLDB is "fast and much more memory efficient than GDB at loading symbols" [8].

libc++ and libc++ ABI

libc++ is an implementation of the standard C++ library for C++11 and higher. It was designed for fast execution and low memory use and ensuring correctness defined by C++ standard. Although there exist many standard libraries for C++, LLVM decided to build up its own. The reason behind it was, that to increase the library qualitatively, and fundamental changes to their way of implementing the libraries would be necessary [8].
ABI stands for Application Binary Interface. It describes how the generated object file is executed by the processor. This means the ABI is platform dependent. When compiling the source code, the compiler has to be aware of the ABI of this specific processor. Therefore, the same object file can't be executed on different computers directly. And

if the ABI of a processor changes, an object file which was compiled for this specific processor can't be executed any more as well.
Therefore, LLVM created a standard C++ library in a more effective way when looking at run and execution time and memory use and developed the libc++ ABI library for the low level support for a standard C++ library.

compiler-rt

Compiler-rt are runtime libraries. They can be divided into four groups: builtins, sanitizer runtimes, profile and and BlocksRuntime [8].
Builtins provides an optimized implementation of the low-level target-specific hooks required by code generation and other runtime components. The implementations are in target independent C form or in a heavily-optimized assembly form. Builtins partly supports libgcc interfaces and further routines can be added if needed.
Sanitizer runtimes are libraries needed to detect errors in the code while running in exchange for a slower compilation. The AdressSanitizer library is a fast memory error detector. It can be used to find for example out of bounds accesses for a slowdown of 2x. The ThreadSanitizer detects data races in exchange for a slowdown of 5x-15x and a memory overhead of 5x-10x. The UndefinedBehaviourSanitzizer library can be used to check for undefined behaviour and the LeakSanitizer for memory leak detection. The MemorySanitizer detects uninitilized reads for a slowdown of 3x. Finally the DataFlowSanitizer provides a data flow analysis framework the programmer can use to find application-specific problems.
Profile is used to collect coverage information and BlockRuntime is a target independent implementation of the Apple runtime interface "Blocks".
Compiler-rt is used mainly by Clang and other LLVM projects.

MLIR

MLIR probably is the newest project of LLVM. It stands for multi-level intermediate representation and aims to provide a framework for defining intermediate representation to significantly reduce the cost to build domain specific compiler and support multiple different requirements in a unified infrastructure [8]. One requirement on MLIR is, that it should be able to represent dataflow graphs and optimisations and transformations on it.

OpenMP

OpenMP contains the required components to build an executable OpenMP program outside the compiler. OpenMP stands for open multi-processing and is an Application Programming Interface (API) for shared memory multiprocessing. It splits up the code on several threads to reduce the runtime [8].
API are how callable functions of libraries are defined so a programmer can use them by using a defined command.

polly

Polly is a data-locality optimizer. It optimizes the memory access pattern of a program by using an abstract mathematical representation of a polyhedral model. The optimisations are mostly done by loop tiling, which exploits spacial and temporal locality of data accesses in loop nests so data can be accessed in blocks and loop fusion.

libclc

Libclc is a library with necessary routines to run the OpenCL (Open Computing Language) C programming language. To increase the performance of computers the technique used also changed a lot. CPUs (Central Processing Unit) improved by adding multiple cores, while GPUs (Graphics Processing Unit) changed into programmable parallel processors. OpenCL aims to get software developers access to this advantages [14].
Libclc is supposed to be used with the Clang OpenCL Front end.

KLEE

KLEE is the implementation of a "symbolic virtual machine", designed to find bugs and even to create a testcase for the bug.
In general, bugs such as functional correctness bugs are difficult to find without execution of the code. But with the help of symbolic execution and automatically created test input they can be found. The test input is initially completely arbitrary. The symbolic machine replaces operation in the code with ones that just generate symbolic values. If the code hits a branch a set of constraints is created for each path, the path conditions [15].
In case of KLEE, when a path hits a bug a testcase can automatically be generated by replacing the symbolic values of the path conditions with real values.

LLD

LLD is a linker project of the LLVM. Linkers are necessary to generate an executable file, a library file or another object file from one or more object files created by the Back end. LLD consist of several different linkers and is designed as an drop-in replacement for system linkers and runs much faster than them.
LLVM also gives an comparison for the runtime advantage: "When you link a large program on a multi-core machine, you can expect that LLD runs more than twice as fast as the GNU gold linker." [8]

3.2 Conclusion

LLVM is a open source compiler project, which was able to become one of the largest compiler project in a short period of time to due to a modular and library-based structure, which allows easy changes fitting the specific requirements of each programmer and a lot of documentations and possibilities for support.
Many sub-projects are not completed yet or get extended for more platforms and to fit the programmers needs even more. When working a lot on compilers and runtime and memory use are of large interest, it is probably worth it to take a look into LLVM.

References

1. https://www.guru99.com/compiler-design-lexical-analysis.html [17.02.2020]
2. https://www.guru99.com/syntax-analysis-parsing-types.html [17.02.2020]
3. https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/ [17.02.2020]
4. https://medium.com/@ChrisCanCompute/how-do-compilers-work-2-middle-end-c4c8cff80b90 [17.02.2020]
5. https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture [17.02.2020]
6. https://www.aosabook.org/en/llvm.html#fig.llvm.rtc
7. https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf [27.02.2020]
8. https://llvm.org/ [27.02.2020]
9. https://queue.acm.org/detail.cfm?id=2544374 [27.02.2020]
10. https://courses.engr.illinois.edu/cs426/fa2019/Notes/5ir.4up.pdf [27.02.2020]
11. http://www.gnu.org/gnu/thegnuproject.en.html#content [27.02.2020]
12. https://medium.com/@alitech_2017/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378 [27.02.2020]
13. https://awards.acm.org/award_winners/lattner_5074762 [27.02.2020]
14. https://www.khronos.org/registry/OpenCL/specs/opencl-1.1.pdf
15. https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf