Eötvös Loránd Tudományegyetem
Faculty of Informatics
Department of Programming Languages And Compilers

Improving Performance of C++ Programs with Static Analysis

Gábor Horváth
Computer Science MSc

Barnabás Bittner
Software Engineering BSc

Budapest, 2017

Contents

List of Figures

1 Introduction

Static analysis is the analysis of computer software without the need to fully compile and execute it as opposed to dynamic analysis[1] which involves running the application. With the help of static analysis we can uncover hard-to-find bugs in the codebase or we can spot potential inefficiencies which otherwise would be hidden to the developers.

2 Technical background

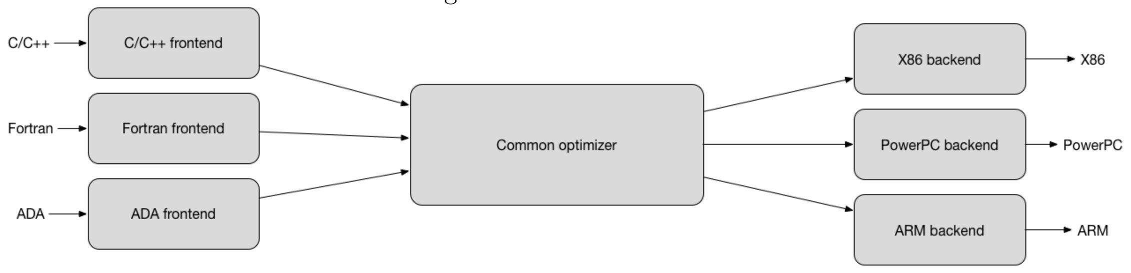
In this section I will outline different technologies related to the main goal of this thesis. Starting from the global framework, which I'm integrating modules into, to the abstract representation of a C++ application. Also I will highligth the importance of every module to the global picture.

2.1 LLVM

LLVM formerly known as Low Level Virtual Machine is a "collection of the reusable and modular compiler technologies"[2]. LLVM started out as a university project[3], and since then it grew significantly in size and it now offers numerous subprojects which help building and maintining both commercial and open-source applications.

The essential goal of LLVM is to provide generalized optimizations to arbitrary programming languages using the LLVM Intermediate Language also known as LLVM IR, which acts as a common representation of different programming languages. This is achieved through using specific language front-ends, which transforms the given language to LLVM IR.

Figure 1: LLVM workflow



A common use-case of LLVM begins with an aforementioned language font-end, which will tranform a given language, C++ in our case, to LLVM IR. The common optimizer, then will perform certain optimizations with this intermediate representation depending on the various settings, for instance if we want smaller or faster code. After the optimizer did its job it will transfer the optimized IR to a certain back-end, again depending on different settings, which will generate the actual executable code for the specified architecture.

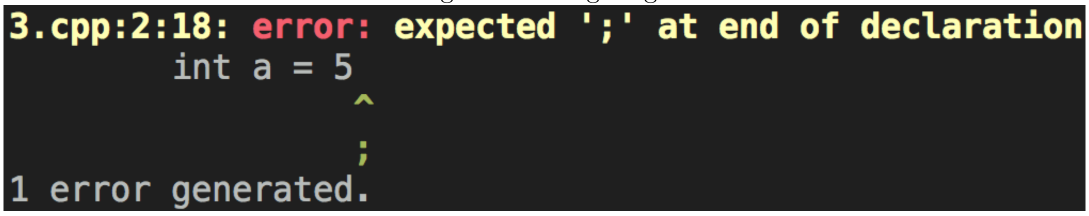
Apart from defining a generalized way of code optimization LLVM provides a full framework of other utility classes and tools, with which one can write better, faster code.

2.2 Clang

The most widely spread C/C++ familiy compiler front-end for LLVM is Clang which aims to excel from the open-source compilers with its exceptionally fast compile-times and user-friendly diagnostic messages[4]. Because of its library oriented design it's really easy to integrate new modules into the Clang ecosystem and also it can be scaled much more effectively than a monolithic system.

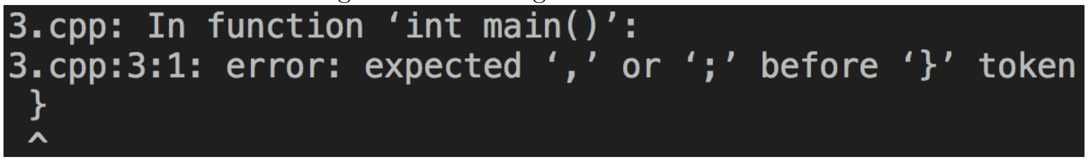
Clang tries to be as user-friendly as possible with its expressive diagnostic messages emitted during compilation. This includes for instance printing the exact location where the erronous code is, displaying a caret icon (ˆ) at the exact spot. Also Clang's output is colored by default making it easier to see what the problem is. Also Clang can represent intervals in the output to show which segment of the code needs changing, along with so-called FixItHints which are little modifications to the code required to fix a certain problem. These can include inserting new code, removing old, and modifying existing.

Figure 2: Clang diagnostic



```
3.cpp:2:18: error: expected ';' at end of declaration
        int a = 5
                 ^
                 ;
1 error generated.
```

Figure 3: GCC diagnostic on the same error



```
3.cpp: In function ‘int main()’:
3.cpp:3:1: error: expected ‘,’ or ‘;’ before ‘}’ token
 }
 ^
```

Clang currently supports the vast majority of the newest features of the in-progress C++17 Standard, and fully supports C++98, C++11 and C++14[5].

2.3 Clang-tidy

2.4 Abstract syntax tree - AST

3 User documentation

3.1 Installing the program

3.1.1 Windows

3.1.2 Linux

3.1.3 OS X

3.2 Using the program

4 Developer documentation

5 Testing

6 References

[1] D. C. L. W. N. W. BA Wichmann, AA. Canning and D. Marsh, “Industrial perspective on static analysis,” *Software Engineering Journal*, 1995.

[2] “The llvm compiler infrastructure,” 2017. April. `http://llvm.org/`.

[3] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, (Palo Alto, California), Mar 2004.

[4] “Clang - features and goals,” 2017. April. `http://clang.llvm.org/features.html`.

[5] “C++ support in clang,” 2017-03-16. `http://clang.llvm.org/cxx_status.html`.