



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
1 Introduction	2
2 Technical background	3
2.1 LLVM	3
2.2 Clang	3
2.3 Clang Static Analyzer	4
2.4 Clang-tidy	4
2.5 Abstract syntax tree - AST	4
3 User documentation	7
3.1 Problems solved by new modules	7
3.1.1 Inefficient String Concatenation[7]	7
3.2 Installing the program	7
3.2.1 Windows	7
3.2.2 Linux	7
3.2.3 OS X	7
3.3 Using the program	7
4 Developer documentation	8
4.1 Detailed description of the solved problems	8
4.1.1 Inefficient String Concatenation Check	8
4.1.2 Inefficient Stream Use Check	8
4.1.3 Default Container Initialization Check	8
4.1.4 Polymorphic Shared Pointer Cehck	8
4.2 Implementations	8
4.2.1 Inefficient String Concatenation Check	8
4.2.2 Inefficient Stream Use Check	8
4.2.3 Default Container Initialization Check	8
4.2.4 Polymorphic Shared Pointer Cehck	8
5 Testing	9
6 References	10

List of Figures

1 LLVM workflow	3
2 Clang diagnostic	4
3 GCC diagnostic on the same error	4
4 AST represented as a tree graph	6

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.

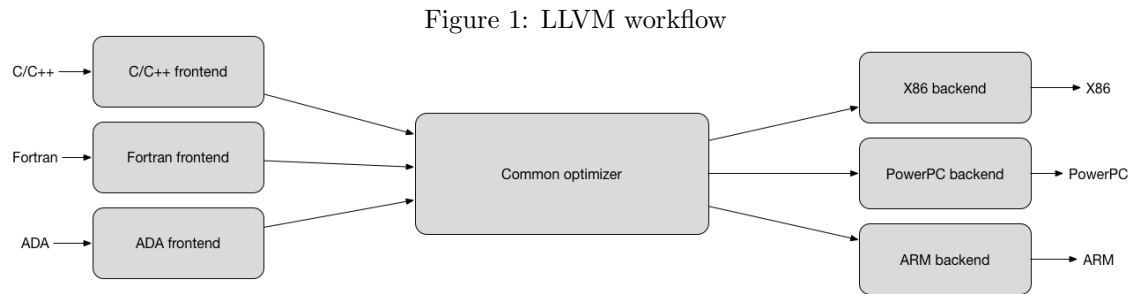
2 Technical background

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 highlight 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 maintaining 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.



A common use-case of LLVM begins with an aforementioned language front-end, which will transform 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.

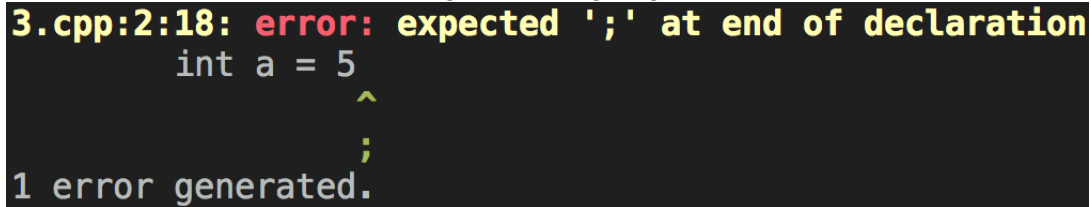
2.2 Clang

The most widely spread C/C++ family 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 erroneous code is, displaying a caret icon (^) at the exact spot. Also Clang's output is coloured by default making it easier to see what the problem is. Also Clang can represent intervals in the output to show

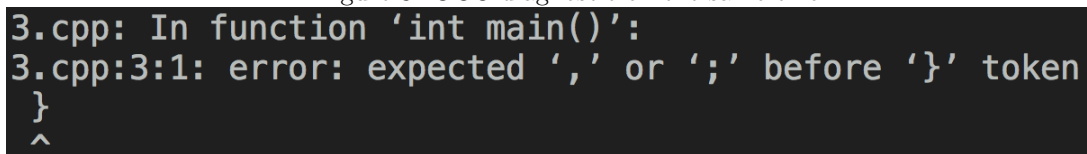
which segment of the code needs changing, along with so-called `FixItHints` which are little modifications to the code required to fix a certain problem. These can include inserting new code, removing old, and modifying existing.

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 Static Analyzer

This project is built on top of Clang and LLVM and it's a tool for finding bugs in C, C++ and Objective-C code with the use of static analysis. This is a stand alone tool which can be called from the command line and it consists of several modules each checking for different types of bugs in the code.

2.4 Clang-tidy

Clang-tidy is a very similar tool to Clang Static Analyzer but it extends its functionality providing more modules while also being able to run the its checks. It defines itself as "a clang-based C++ "linter" tool"[6]. It uses Clang's expressive diagnostics to print out warnings to the user and also provides support for `FixItHints` which can be used to automatically fix and refactor erroneous source code.

In this thesis I am creating four different modules for Clang-tidy which can potentially improve the execution time of C++ applications. Each module addresses different aspects of coding errors which can be rewritten in a more effective way.

2.5 Abstract syntax tree - AST

The fundamental bedrock of compiling a programming language is the construction of the abstract syntax tree, which is a tree that represents language specific constructs annotated with all the information what's needed to generate machine code from the source. From now on when I am talking about AST I mean specifically an AST generated from C++ code.

The AST doesn't contain every syntactic information from the source code, for instance it will not contain semicolons after statements. But will include information about implicit casting of

variables, implicit constructor calls, temporary object creation and many other implicitly coded information.

```
1 int main()
2 {
3     int a = 5;
4     int b = a + 4;
5     return b;
6 }
```

Listing 1: A simple C++ program

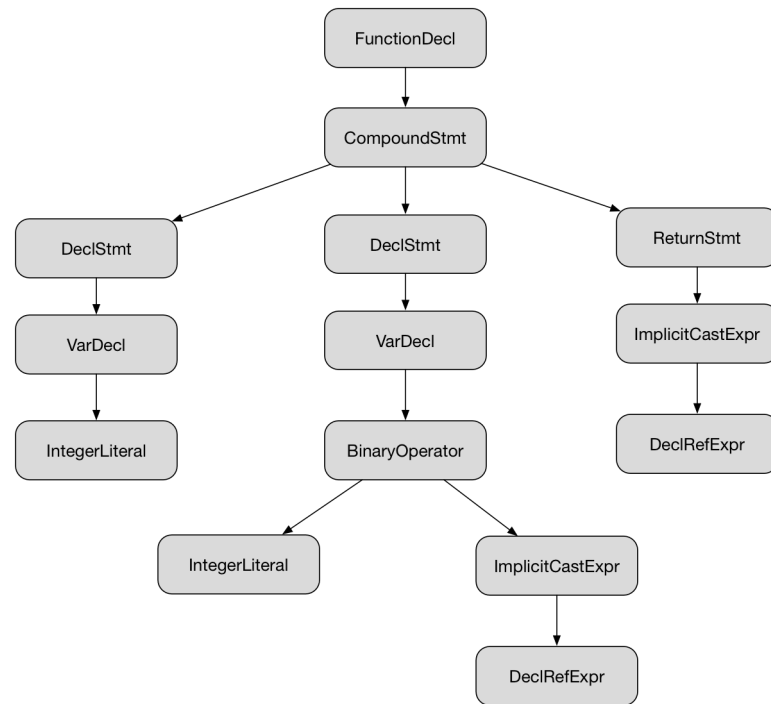
Comparing a simple program in Listing:1 with the AST generated from it in Listing:2 we can observe all the extra information the AST has. We can see implicit casting from lvalue to rvalue, a node for expressing compound statement, a lot of memory addresses, source locations and other meta information about the code.

```
1 FunctionDecl 0x7fe488 <basic.cpp:1:1, line:5:1> line:1:5 main 'int (void) '
2   `--CompoundStmt 0x7f15c8 <col:12, line:5:1>
3     |--DeclStmt 0x7fe48d <line:2:3, col:12>
4       | `--VarDecl 0x7fe480 <col:3, col:11> col:7 used a 'int' cinit
5         |   `--IntegerLiteral 0x7f1420 <col:11> 'int' 5
6       |--DeclStmt 0x7fe485 <line:3:3, col:16>
7         | `--VarDecl 0x7fe48 <col:3, col:15> col:7 used b 'int' cinit
8           |   `--BinaryOperator 0x7fe48d <col:11, col:15> 'int' '+'
9             |   |--ImplicitCastExpr 0x7fe48d <col:11> 'int' <LValueToRValue>
10              |   |   `--DeclRefExpr 0x7fe48d <col:11> 'int' lvalue Var 0x7fe480 'a' 'int'
11              |   |   `--IntegerLiteral 0x7fe4f8 <col:15> 'int' 4
12           `--ReturnStmt 0x7fe48d <line:4:3, col:10>
13             `--ImplicitCastExpr 0x7fe598 <col:10> 'int' <LValueToRValue>
14               `--DeclRefExpr 0x7fe480 <col:10> 'int' lvalue Var 0x7fe48 'b' 'int'
```

Listing 2: AST generated

The generated AST can also be represented as a tree graph to make it visually more appealing. This model represent the best how the different nodes are connected together.

Figure 4: AST represented as a tree graph



3 User documentation

3.1 Problems solved by new modules

The four new modules created for Clang-tidy aim to address four fairly distinctive and inefficient programming patterns.

3.1.1 Inefficient String Concatenation[7]

The first problem involves the standard library's string class. This class, more precisely `std::basic_string<char>`, is the default implementation of a modifiable high-level character sequence in the language. It provides several functions which can modify, transform the underlying characters.

The problem arises when one wants to concatenate two or more strings. There are several methods achieving the concatenation: `operator+`, `operator+=` and `.append` member function. The problem arises when one wants to include the original string in the concatenation with a structure similar to `a = a + b`; which is highly inefficient. In Listing:4 the concatenation using `operator+` is causing a noticeable performance overhead in the application.

```
1 std::string a("Foo"), b("Baz");
2 for (int i = 0; i < 20000; ++i)
3 {
4     a = a + "Bar" + b;
5 }
```

Listing 3: Highly inefficient code

This program could be refactored into a much faster code using either `operator+=` or `.append()` member function as shown in Listing:??.

```
1 std::string a("Foo"), b("Baz");
2 for (int i = 0; i < 20000; ++i)
3 {
4     a.append("Bar").append(b);
5 }
```

Listing 4: A more efficient version

3.2 Installing the program

3.2.1 Windows

3.2.2 Linux

3.2.3 OS X

3.3 Using the program

4 Developer documentation

4.1 Detailed description of the solved problems

4.1.1 Inefficient String Concatenation Check

4.1.2 Inefficient Stream Use Check

4.1.3 Default Container Initialization Check

4.1.4 Polymorphic Shared Pointer Cehck

4.2 Implementations

4.2.1 Inefficient String Concatenation Check

4.2.2 Inefficient Stream Use Check

4.2.3 Default Container Initialization Check

4.2.4 Polymorphic Shared Pointer Cehck

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.
- [6] “Clang-tidy,” 2017. <http://clang.llvm.org/extra/clang-tidy/index.html>.
- [7] “Extra clang tools 5 documentation,” 2017. <https://clang.llvm.org/extra/clang-tidy/checks/performance-inefficient-string-concatenation.html>.