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[2] 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

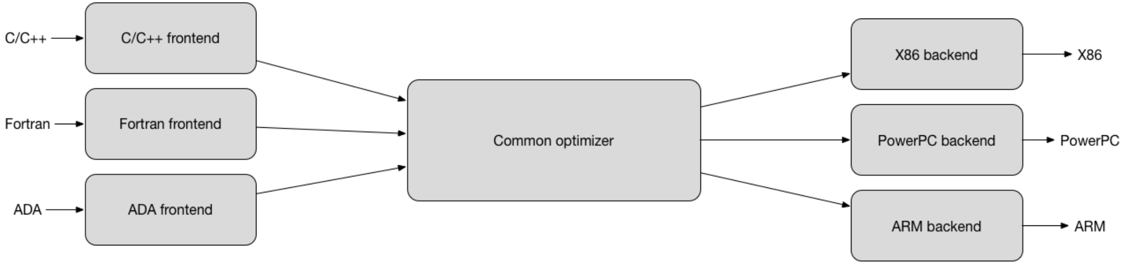
In this section I will outline different technologies related to the main goal of this thesis. Starting from the global framework, which I'm integrating modules into, to the abstract representation of a C++ application. Also I will highligth the importance of every module to the global picture.

## 2.1 LLVM

LLVM formerly known as Low Level Virtual Machine is a "collection of the reusable and modular compiler technologies"[3]. LLVM started out as a university project[4], 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.

Figure 1: LLVM workflow



A common use-case of LLVM begins with an aforementioned language fontend, 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.

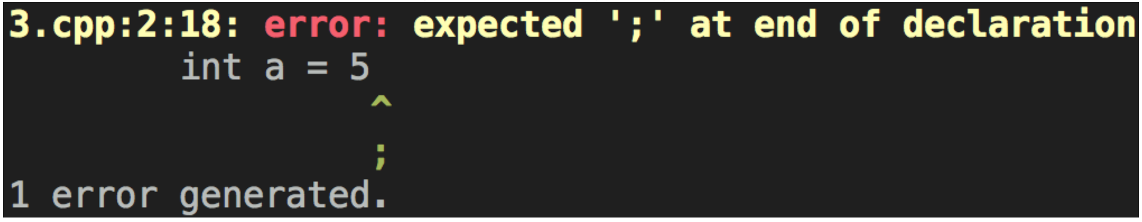
Apart from defining a generalized way of code optimization LLVM provides a full framework of other utility classes and tools, with which one can write better, faster code.

## 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[5]. 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.

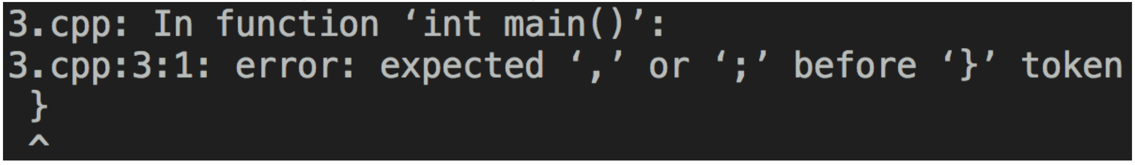
Clang tries to be as user-friendly as possible with its expressive diagnostic messages emitted during compilation. This includes for instance printing the exact location where the erroneous code is, displaying a caret icon (ˆ) at the exact spot. Also Clang's output is coloured by default making it easier 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[6].

## 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”[7]. 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.

Clang-tidy also provides access to all the Clang and LLVM framework, making it easy to navigate the abstract syntax tree generated from the C++ code or reading the source-file’s text, using the parser to get the next token from a given location for example and also the semantic analyzer functionalities.

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 a so-called 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.

```cpp
int main()
{
    int a = 5;
    int b = a + 4;
    return b;
}
```

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.

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

In Clang the AST nodes are implemented as C++ classes with the proper use of inheritance, so the whole AST follows an object-oriented design, making it much easier to work with. In static analysis, we are primarily try to find the patterns connected to the problematic code, and then traverse the AST looking for these patterns.

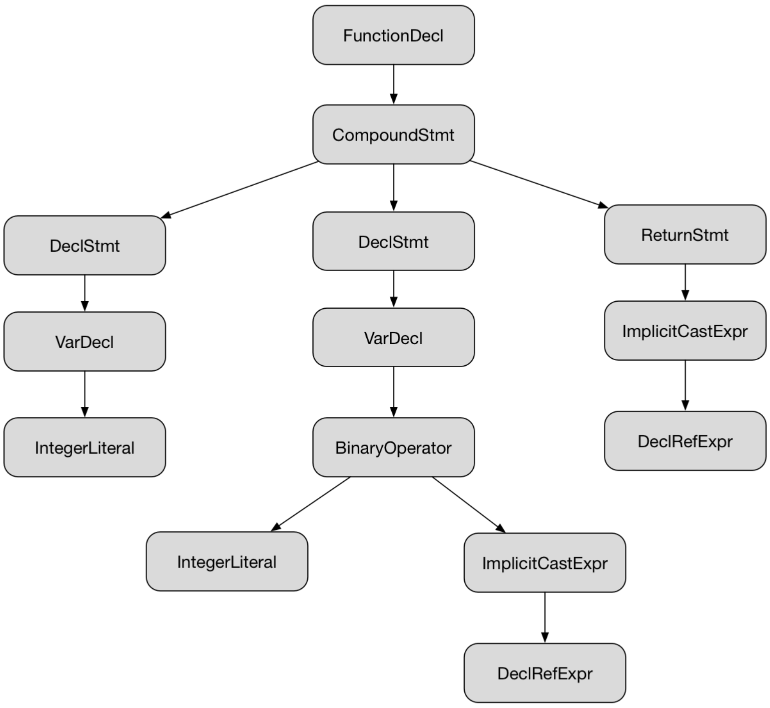
Clang provides numerous so-called AST matchers[8], which are structures that can match different predicates on AST nodes. This greatly simplifies how one can traverse and look for a specific code pattern in the tree.

```
varDecl(matchesName("foo.*"))
forStmt(hasDescendant(varDecl()))
declRefExpr(hasDeclaration(varDecl(hasName("foo"))))
```

Listing 3: Basic AST matchers

In Listing:3 there are some examples of AST matchers. From the first line descending, a matcher that matches a variable declaration whose name begins with "foo"; for statements which have at least one variable declaration in their body; a "foo" named variable used in some context.

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 fairy distinctive and inefficient programming patterns.

### 3.1.1 Inefficient String Concatenation[1]

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.

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

Listing 4: Highly inefficient code

This program could be refactored into a much faster code using either `operator+=` or `.append()` member function as shown in Listing:5.

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

Listing 5: A more efficient version

This check doesn't involve a FixItHint because the solution cannot be generalized.

### 3.1.2 Inefficient Stream Use

The second problem involves the standard library's `operator<<` and `std::ostream`. In C++ you can use `std::cout` to print to the console output, which is really helpful for logging or just this is the way the program communicates with the user, for instance Clang-tidy or Clang itself.

There are two issues with using the standard way of interaction with the console. First is when using single characters yet annotating them with `""` (double quotes), for example: `"a"`. It's quite inefficient when you put such constructed characters to the stream. Instead one should construct single characters as `'a'`.

```
std::cout << "a" << "b" << "c";
```

Listing 6: Sightly inefficient way of streaming characters

```
std::cout << 'a' << 'b' << 'c';
```

Listing 7: A generally more efficient version

The other problem which raises more concern about the performance is streaming multiple `std::endl` after each other. Unnecessary use of `std::endl` can potentially cause massive performance hogs in an application especially if it's a library which is intended to be used by others. For example the code in Listing:8 can be rewritten in a highly more efficient form as in Figure:9

```
std::cout << std::endl << std::endl << std::endl;
```

Listing 8: A really slow way to print newlines

```
std::cout << '\n' << '\n' << std::endl;
```

Listing 9: A more efficient version

This check contains FixItHints for correcting the aforementioned issue. It replaces `""` with `''` and also rewrites multiple `std::endl` function calls in a stream, except for the last one.

### 3.1.3 Shared Pointer Conversion

This problem is relevant to `std::shared_ptr` in the standard library. A `shared_ptr` is a smart pointer, therefore it cannot dangle (point to invalid memory address), which allows other `shared_ptrs` to point at the same object as the first pointer, as opposed to `std::unique_ptr` which doesn't allow multiple owners of the object it points to. This makes it slightly more overweight than the `unique_ptr` because each pointer has to know whether it's the last one keeping a reference to the object, and if so when it's destructed it needs to free the resources allocated for the object it points to.

In C++ where classes are involved with the use of polymorphism a derived class' instance can always be converted to a base class' one. With this casting operation and `shared_ptr`'s way of knowing how many references there are for the object, passing `shared_ptrs` via function arguments where there is implicit (or explicit) casting used is really inefficient in terms of performance.

```
class A {};
class B : A {};
void f(std::shared_ptr<A>){}
int main()
{
    auto ptr = std::make_shared<B>();
    f(ptr);
}
```

Listing 10: Inefficient implicit cast

In Listing:10 there is a small example program demonstrating the code which can be greatly improved just by getting the reference of the underlying object of the pointer, as seen in Listing:11. This program is about 2.5 times faster than the one before.

```
class A {};
class B : A {};
void f(const A&){}
int main()
{
    auto ptr = std::make_shared<B>();
    f(*ptr.get());
}
```

Listing 11: A much faster version

For this check there's no FixItHint option, because there are different ways of refactoring the inefficient structure and if the user accessed the `shared_ptr`'s API then it would be nearly impossible to find a simple automatic solution for refactoring such code.

### 3.1.4 Default Container Initialization

The last problematic code pattern addressed involves the standard library's default containers, namely `std::vector`, `std::set`, `std::deque` and `std::map`. In C++ when initializing containers one can initialize them and add the elements afterwards, which is slightly less efficient than initializing the containers in the expression they are declared.

```
std::vector<int> vec;
vec.push_back(3);
vec.emplace_back(1.2);
vec.push_back(1.0);
```

Listing 12: Inefficient way of creating containers

As in Listing:12 the initialization of an `std::vector` could be done in a more efficient way, with less lines of code. In Listing:13 the refactored version needs some explicit conversions to work.

```
std::vector<int> vec{3, (int) 1.2, (int) 1.0};
```

Listing 13: A faster way of initializing

## 3.2 Installing the program

Installing the program with the new modules is fairly an easy process, but it varies from operating system to operating system.

### 3.2.1 Windows

Installing the tool on Microsoft Windows is a little bit more complicated than on Linux. There are several paths that you can take to set up a Linux environment inside Windows or you can use a native Windows approach.

#### 3.2.1.1 MinGW

#### 3.2.1.2 Cygwin

#### 3.2.1.3 WSL

### 3.2.2 OS X

On macOs (formerly OS X) the installation is fairly straightforward, we just need some preparation before we can continue with the Linux guide in 3.2.3. First of all because macOs is a Unix type system (its roots are in BSD) working in a Terminal on macOs is quite familiar experience with a native Linux one, both having Bash as the default terminal application. But before we can deep dive into the Linux steps we need to install a package manager, because macOs doesn't have one out of the box.

We will install the Homebrew[9] package manager, as this supports the most features and it's the most supported one. Open a terminal and type:

```
user@host:$ ruby -e "$(curl -fsSL https://raw.
    githubusercontent.com/Homebrew/install/master/install)
    "
```

This will install the whole homebrew system, and after that you will be able to install packages with the `brew install` command. Now you should install Xcode from the AppStore as it will install a lot of useful packages to the system, for instance `gcc`, `ld`, `clang`.

Now we should install the requirements to download LLVM and Clang:

```
user@host:$ brew install svn, cmake
```

After successfully installing these applications you can head to the second point of the Linux guide to finalize the installation process.

### 3.2.3 Linux

Before we start installing the tool on a Linux system, we have to install some basic applications needed to download and compile the program. In the Linux world there are a lot of package manager applications, for instance there is (on default) `apt-get` on Debian-based systems, `Pacman` on Arch Linux-based systems, `rpm` on RedHat-based systems and `Portage` on Gentoo-based systems. To install a package on these different systems use the following commands (supposing one have superuser privileges):

```
user@host:$ apt-get install <package-name>
user@host:$ pacman -S <package-name>
user@host:$ yum install <package-name>
user@host:$ emerge <package-name>
```

From now on, I will show the commands only on Debian-based systems, but translating the given instructions to a different Linux flavour shouldn't be hard.

Firstly we have to install subversion(`svn`) on the machine which is a version control system. Also we will need `cmake` which can generate Linux makefiles to make it easy to compile the application. To install them, type:

```
user@host:$ sudo apt-get install svn, cmake
```

We need subversion because the LLVM main project and Clang is in an svn repository over the internet. Then we need to clone (download) LLVM first, then clone Clang into the previously cloned LLVM project. Navigate to a comfortable directory where you want your project to reside and clone the repositories:

```
user@host:$ cd /directory/where/your/project/should/be
user@host:$ svn co http://llvm.org/svn/llvm-project/llvm/
   trunk llvm
user@host:$ cd llvm/tools
user@host:$ svn co http://llvm.org/svn/llvm-project/cfe/
   trunk clang
user@host:$ cd ../..
```

If you are comfortable with the default C++ standard library you can skip the next step, otherwise if you want to use newer libc++ standard library implementation, proceed with the following:

```
user@host:$ cd llvm/projects
user@host:$ svn co http://llvm.org/svn/llvm-project/
   libcxx/trunk libcxx
user@host:$ cd ../..
```

Now you have the framework required for the `extra` module. Copy the `extra` directory from the location where the thesis is, to the Clang project:

```
user@host:$ cp /dir/to/thesis/location/extra llvm/clang/
   tools/
```

Now we are ready to compile the whole project. Create a build directory outside of the project structure (in-tree builds are not supported[10]), then generate Unix Makefiles and compile the project. We pass `-DCMAKE_BUILD_TYPE=RELEASE` argument to `cmake` to set it to release build type.

```
user@host:$ mkdir build
user@host:$ cd build
user@host:$ cmake -G "Unix Makefiles" ../llvm -
   DCMAKE_BUILD_TYPE=RELEASE
user@host:$ make
```

Cmake supports many modern IDEs, see its documentation for different generators[11].

After the whole project compiled successfully you will find the built executable files in `build/bin` directory.

## 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] "Extra clang tools 5 documentation," 2017. https://clang.llvm.org/extra/clang-tidy/checks/performance-inefficient-string-concatenation.html.

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

[3] "The llvm compiler infrastructure," 2017. April. http://llvm.org/.

[4] 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.

[5] "Clang - features and goals," 2017. April. http://clang.llvm.org/features.html.

[6] "C++ support in clang," 2017-03-16. http://clang.llvm.org/cxx_status.html.

[7] "Clang-tidy," 2017. http://clang.llvm.org/extra/clang-tidy/index.html.

[8] "Ast matcher reference," 2017. http://clang.llvm.org/docs/LibASTMatchersReference.html.

[9] "Homebrew," 2017. https://brew.sh/.

[10] "Getting started: Building and running clang," 2017. https://clang.llvm.org/get_started.html.

[11] "cmake-generators," 2016. https://cmake.org/cmake/help/v3.5/manual/cmake-generators.7.html.