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 and we can spot potential inefficiencies which otherwise would be hidden to the developers.

It complements traditional testing by providing tools to check such aspects of software which could not be done by traditional testing. Testing the performance of an application with automated tests are harder than with the help of static analysis. Static analysis can also be used to check for violations of coding conventions and other mechanical and repetitive tasks which could be done by people but can be effectively automatised to cut costs and reduce the possibility of error.

Static analysis also can be used to perform automatic source-to-source transformation of source code.

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"[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 to build and maintain both commercial and open-source applications.

The essential goal of LLVM is to provide generalised optimisations 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
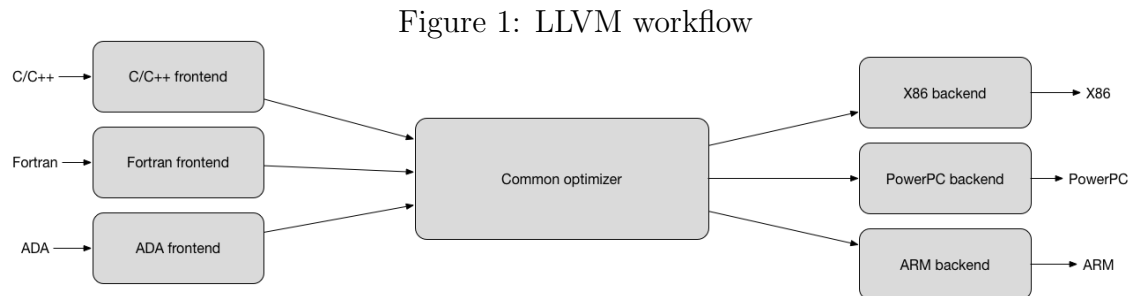
A common use-case of LLVM begins with an aforementioned language fontend, which will transform a given language, C++ in our case, to LLVM IR. The common optimiser then will perform certain optimisations with this intermediate representation depending on the various settings, for instance, if we want smaller or faster code. After the optimiser has done its job it will transfer the optimised IR to a certain back-end, again depending on different settings which will generate the actual executable code for the specified architecture.

Describing the LLVM Intermediate Representation it raises the possibility of analysing directly this representation instead of Abstract Syntax Tree. The drawbacks of this proposal are that this representation is too abstract to bear a strong resemblance with the original code which makes it really hard to refactor the original code.

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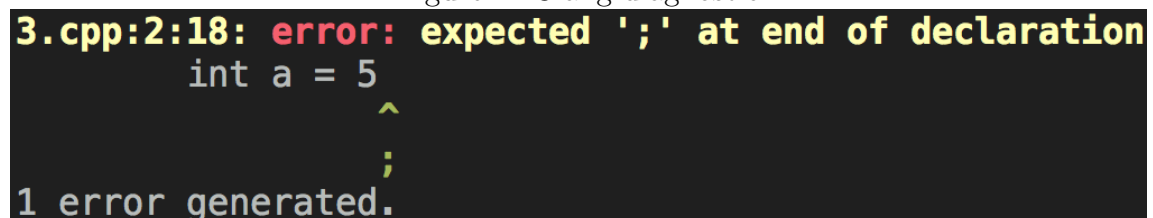
2.2   Clang

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 the Clang ecosystem's library oriented design, it's really easy to integrate new modules into it and also it can be scaled much more effectively than a large 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.
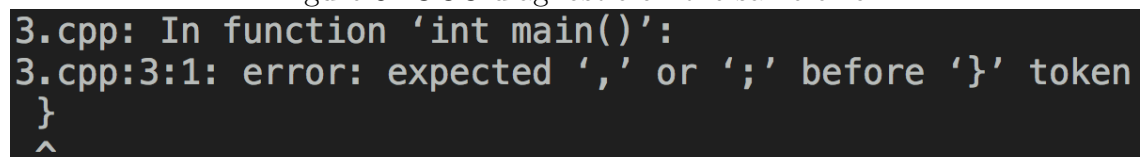
Also Clang's output is coloured by default making it easier to see what the problem is. It can represent intervals in the output to show which segment of the code needs changing, along with so-called FixItHints which are little modifications to the code required to fix a certain problem. These can include 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 similar tool to Clang Static Analyzer in its purpose but it uses the Abstract Syntax Tree to find bugs while Clang Static Analyzer uses symbolic execution to find bugs. It extends its functionality providing more modules while also being able to run the checks of Clang Static Analyzer.

It defines itself as "a clang-based C++ "linter" tool"[7]. It uses Clang's expressive diagnostics to print out warnings for 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 analyser 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 the 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 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 trying 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.

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

Listing 3: Basic AST matchers

Figure 4: AST represented as a tree graph
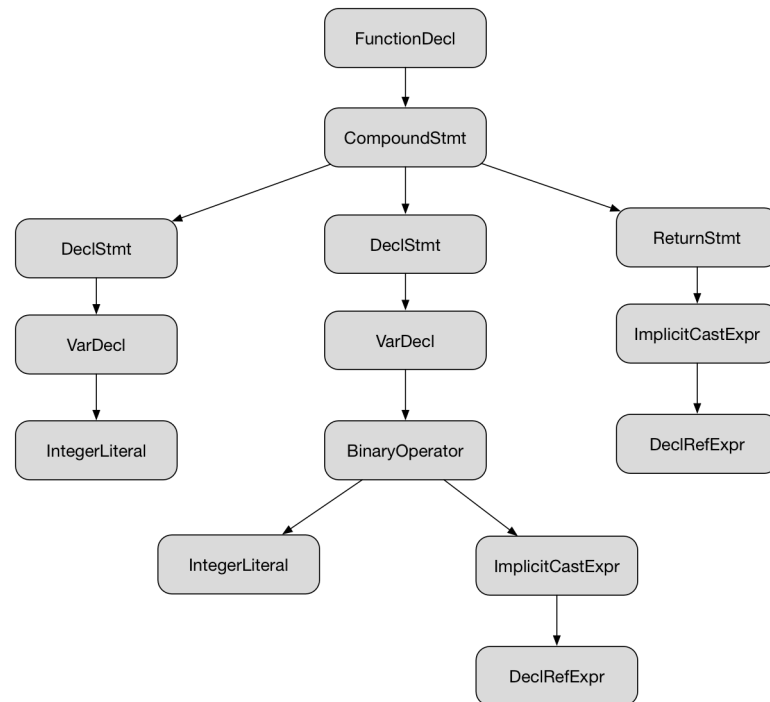
In Listing:3 there are some examples of AST matchers. From the first line descending, a matcher that matches a variable declaration whose name begins with "foo"; for statements which have at least one variable declaration in their body; a "foo" named variable used in some context.

2.6   CMake

CMake is a free, open-source and more importantly cross-platform framework for managing complex C/C++ and Fortran projects. CMake handles everything from building the application itself, testing it and even installing/packaging. It's globally used in the industry, some notable applications that are using it are Netflix, Inria and KDE[9].

Generally CMake workflow is a two-step procedure. Firstly one generates the platform-specific makefiles with CMake, optionally providing extra arguments such as custom property values, or the compiler to use for the actual compilation of

the software. With a great selection of CMake generators it's able to generate project files for popular IDEs for instance Visual Studio, Apple's Xcode or even for CodeBlocks[10].

After the platform specific makefiles or project files are written, then the project is opened with the corresponding IDE or simply compile with `make` command on Unix-like systems.

In light of the aforementioned, it's no surprise that the LLVM project uses CMake too. It simplifies the generation of makefiles or project files, makes it really simple to run the unit and regression-tests associated with LLVM and its subprojects. Also, it makes the project modular with conditionally including different subprojects, for instance, the project can be compiled without the clang-tools-extra project, that contains Clang-tidy, but one can add the project by simply putting the directory in its place and refreshing the CMake configuration.

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[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 and transform the underlying characters.

There are several methods achieving the concatenation of two or more
`std::string`s with `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.

```cpp
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.

```cpp
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 generalised

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 example, 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_ptr`s 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 have a slight overhead against 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_ptr`s via function arguments where it is implicit (or explicit) casting used is really inefficient in terms of performance.

```cpp
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.

```cpp
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 initialising containers one can initialise them and add the elements afterwards, which is slightly less efficient than initialising the containers in the expression they are declared.

```cpp
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.

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

Listing 13: A faster way of initialising

3.2 Performance benchmarks

In this section, I am going to present some performance benchmarks of code which are affected by the problems each check fixes and compare these results to the results of the improved, refactored code. Each test is executed in a well-isolated environment to highlight the execution times of only the code we are interested in.

3.2.1 Shared Pointer Conversion

After applying the suggested fixes by the Shared Pointer Conversion checker we see a significant improvement in speed compared to the previous version.

If we try to fit a line over the points with linear regression we get $y = 0.0002632x + 0.0000059$ with $R^2 = 0.98$ for the refactored one and $y = 0.002938x + 0.0075$ with $R^2 = 1$ for the original one. From these line equations, we can see that the original is about 10 times steeper than the refactored one.

Figure 6: String concatenation benchmark
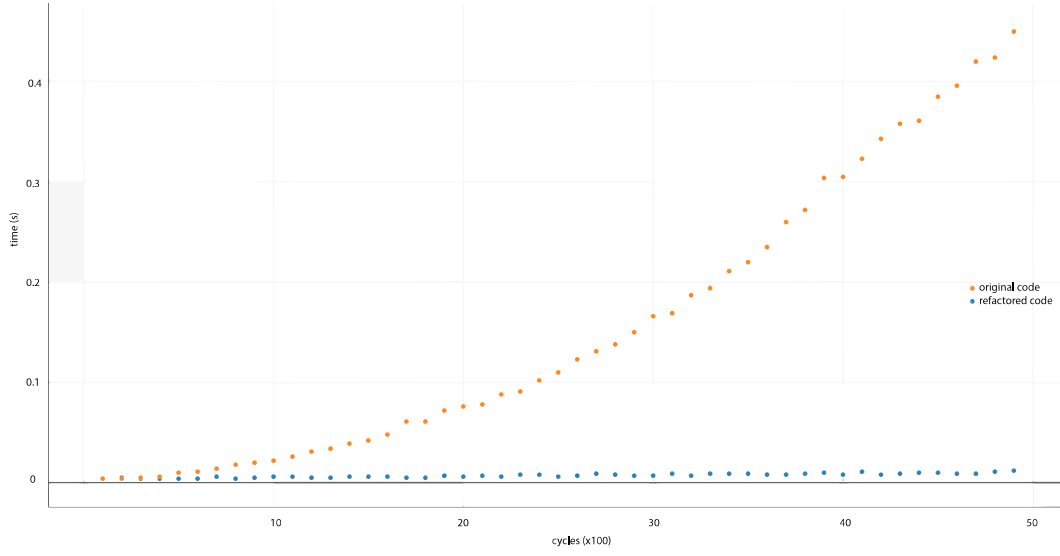
Figure 5: Shared pointer conversion benchmark
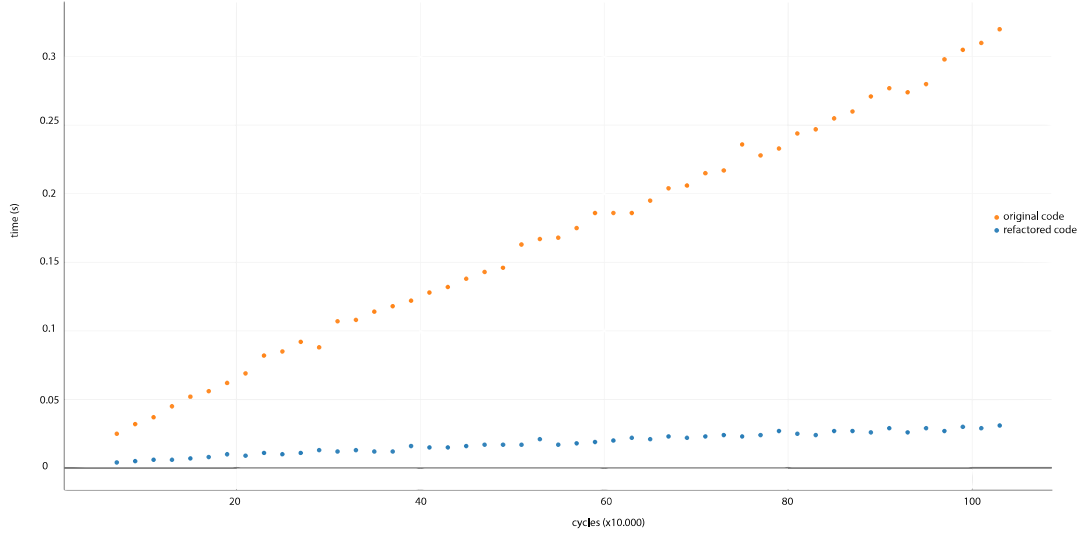
3.2.2   Inefficient String Concatenation

When benchmarking the speed of concatenating strings in a loop the results were surprising. As it turns out naively concatenating strings in a loop is has a quadratic time complexity, yet if we modify on the operation as the check suggests we get a linear time complexity that is much faster than the original version. We can fit a quadratic polynomial $y = 0.0002033x^2 + 0.0086x - 0.00085$ with $R^2 = 1.0$ to the original and a linear $y = 0.0001347x + 0.00380$ with $R^2 = 0.84$ to the refactored.

3.2.3 Container Default Initialization

Benchmarking code that used an inefficient way of initialising an `std::vector` on which the check was run with FixItHints enabled, the automatic refactoring provided a much faster code than the original one. Fitting lines on both samples gives us $y = 0.007635x+0.0038$ with $R^2 = 1.0$ line for the original and $y = 0.002486x+0.0046$ with $R^2 = 1.0$ for the refactored one, resulting in a 3 times faster code.

Figure 7: Container Initialization benchmark
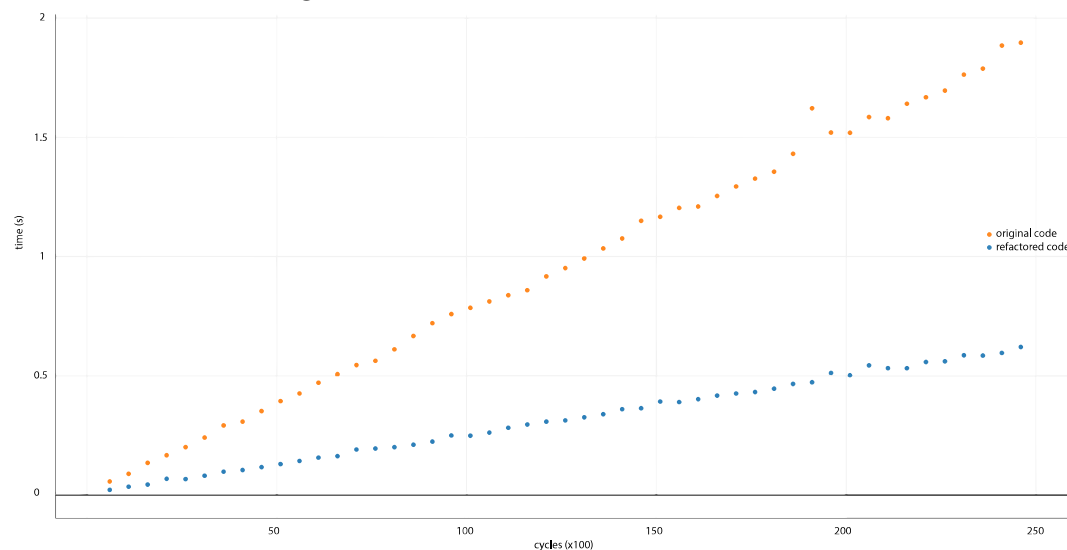
3.2.4 Inefficient Stream Use

Benchmarking this check was divided into two categories because it warns us about two distinct, yet related, performance problems.

First I measured the performance gains of refactoring `<< "a"` type of calls to `<< 'a'` types. The surprising result that I got was that there was no significant increase in performance when applied the FixItHints from the check only a marginal benefit. Performing linear regression analysis on the samples we get $y = 0.004748x+ 0.0061$ with $R^2 = 1.0$ for the original and a slightly smoother $y = 0.004596x + 0.004596$ with $R^2 = 1.0$ for the refactored code.

Figure 8: Char conversion benchmark
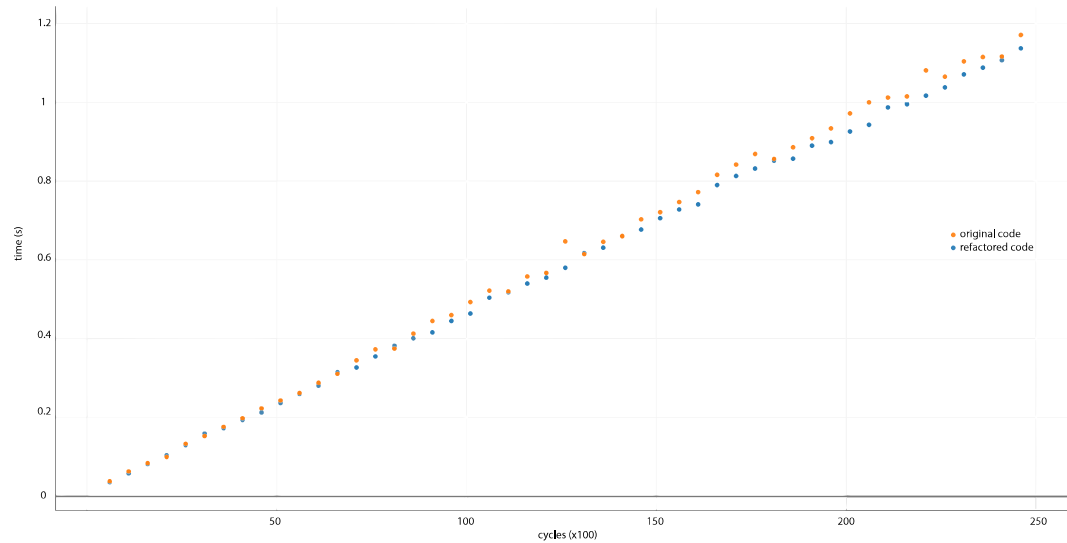
The other type of tests were conducted on using multiple `std::endl` on a single stream. The correction of this problem was also automatic, and the results show 2 times increase in performance after applying the FixItHints. Fitting lines to the original gives a $y = 0.01758x + 0.059$ with $R^2 = 1.0$ and a $y = 0.008256x + 0.001$ with $R^2 = 1.0$ on the refactored.

Figure 9: Multiple end-line benchmark
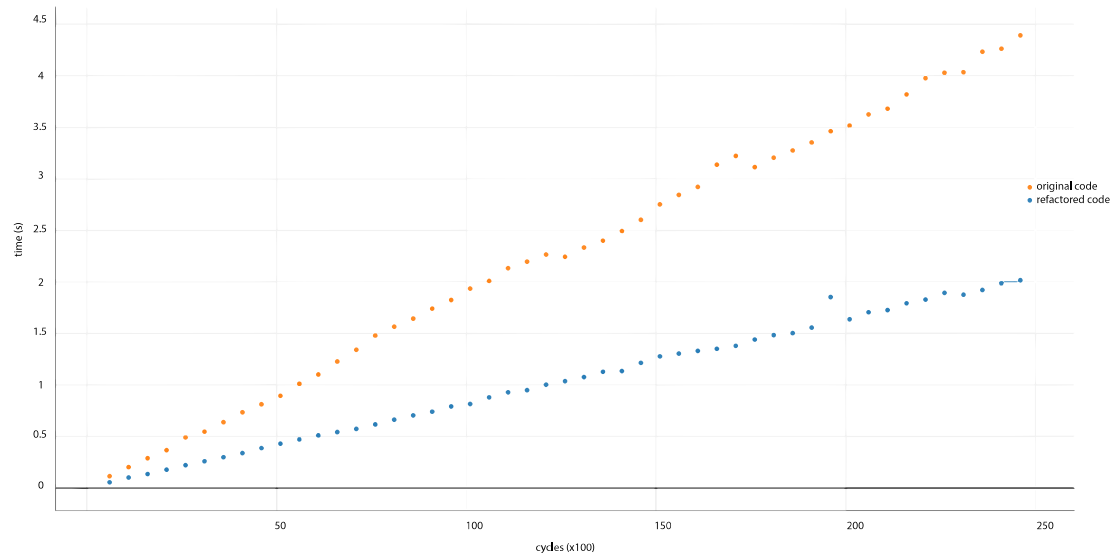
3.3   Installing the program

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

3.3.1   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.

3.3.1.1   MinGW   is a project for Windows which aims to provide a minimalist GNU development environment[11]. This project doesn't aim to provide a POSIX runtime environment[11], but wants to simplify how one can use the C run-time Microsoft provides and other language run-times

MinGW currently features the following applications:

1. A GNU Compiler (`gcc`) port for Windows

2. GNU Binutils which contains a set of utility programs

3. MSYS(Optional)

MSYS is a collection of different utility programs from GNU project. It's an optional complementary designed for MingGW[12]. Its goal is to simplify interacting with UNIX tools, and also to provide a better command-line terminal application for Windows by providing Bash.

To install the program with MinGW, first you have to download the latest MinGW run-time from the original site: `http://www.mingw.org/`. Following this guide one should be able to install MinGW, MSYS, subversion and cmake on their Windows PC: `http://ingar.satgnu.net/devenv/mingw32/base.html`.

With these tools you can continue with the second step of Linux installation guide, following the steps carefully.

3.3.1.2   WSL   also known as Windows Subsystem for Linux is an optional feature from Microsoft on Windows 10 Desktop operating systems. The subsystem enables the execution of native `ELF64` binaries[13] on Windows, without a need of a virtual machine or third party layer.

WSL works by using a so-called Pico process which was developed by Microsoft under project Drawbridge[14] to enable a way of application sandboxing "with minimal kernel API surface"[14]. This Pico process wraps the native Linux application while mapping all the Linux kernel calls to NT kernel calls hence providing the seamless functionality.

To enable WSL in Windows 10, first enable `Developer Mode` in `System Updates` menu. Then open `Windows Features` menu and enable `WSL` and restart the system. After restarting open the program named `Bash on windows` and follow the instructions. After having installed Ubuntu user-mode, proceed to Linux installation guide to complete the installation.

3.3.2   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 3.3.3. 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[15] 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 finalise the installation process.

3.3.3   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[16]), 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[10].

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

3.4   Using the program

After the program has been successfully installed on the given system we can begin using it to run the modules on distinct C++ files or on whole projects.

3.4.1   CLI

When it comes to running the application from the command line, it's important to distinguish running the tool on different, standalone source-files and on a whole C++ project.

Running on standalone source files is the easiest method when only a couple of files are involved which don't require any other library to compile. To run with the default checks enabled on a file named `source.cpp`:

```
user@host:$ clang-tidy ./source.cpp
```

If you want to specify which checks should be used when analysing you can use the `-checks=` flag, and after that, you can use the checker name to add, and a `-` checker name to exclude. You can use `*` to refer to all the checks:

```
user@host:$ clang-tidy -checks=-*,performance-inefficient-stream-
    use ./source.cpp
```

This runs only the Inefficient String Use check. If you need to pass arguments to the compiler behind Clang-tidy you can use the `--` flag and then just pass the arguments. For instance, this code will run Clang-tidy with the default checks enabled, passing `-std=c++14` argument to the compiler:

```
user@host:$ clang-tidy ./source.cpp -- -std=c++14
```

To specifically run the four modules involved in this thesis you can pass these arguments to `-checks` parameter:

- `performance-inefficient-shared-pointer-reference`

- `performance-inefficient-stream-use`

- `performance-inefficient-string-concatenation`

- `performance-container-default-initializer`

There are several options running the tool on a whole project. The fundamental principle to run the tool on multiple files, is to have a JSON Compilation database, which is "a format for specifying how to replay single compilations independently of the build system"[17].

Generating a JSON Compilation database can be done several ways:

- using `cmake` and passing `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON`

- with Ninja build system's `compdb` tool[18]

- since version 4.0.0 Clang supports generating it with `-MJ` argument[19]

3.4.2   GUI

To run Clang-tidy through a GUI is possible with CLion. CLion is a cross-platform C/C++ IDE developed by JetBrains, that is running on JVM. The 2017.2 EAP version of CLion started to support Clang-tidy officially[20]. Apart from the official support, there's a Clang-tidy plugin[21] also which can provide a GUI experience.

Keeping the warnings emitted by Clang-tidy organised and stored for a project can be done by using CodeChecker[22], an open-source project from a collaboration between Ericsson and ELTE. This tool is a "defect database and viewer"[23] that can filter, sort, suppress and visualise warnings produced by either Clang Static Analyzer or Clang-tidy.

4   Developer documentation

4.1   Detailed description of the solved problems

In this section, I am going to present the technical background of the mentioned problems, why they are inefficient and what can we do to refactor them. I am referencing the `N4567`[24] version Working Draft of the C++ standard, which contains every C++14 features and also freely available.

4.1.1   Shared Pointer Conversion Check

According to the C++ standard the `memory` header has to contain a templated class `shared_ptr` with the signature `template<class T> class shared_ptr`[24, §20.8.2.2]

This class is responsible for storing a standard C++ pointer and implementing shared ownership of this raw pointer. This means that the last `shared_ptr` object is responsible for the destruction of the underlying pointer.

The way `shared_ptr` achieves the shared ownership is by a class-wide static variable (`use_count()`) which is incremented in each construction and decremented in each destruction and if `use_count()` reaches zero in a destruction then the underlying pointer shall be destroyed either by default `delete` keyword or by a user-supplied custom deleter. The modification of the reference count has to be an atomic process and should not be modified simultaneously[24, §20.8.2.2 4.].

These properties could make the `shared_ptr` a hidden source of performance hogs when implicit temporary object creation is involved. Such as the case when inherit is used.

The standard says that a pointer to a derived class shall always be implicit convertible to a pointer to the base class[24, §4.10 3.]. Also in C++ objects generated by template instantiations are not covariant, they are invariant in fact, meaning there is no implicit conversion in this example, but a compile time error:

```cpp
struct A {}; struct B : A {};
std::vector<B*> vec1;
std::vector<A*> vec2;
vec2 = vec1;
```

Yet to make `shared_ptr<A>` covariant, to preserve to resemblance to a raw pointer, the standard defines constructors with signature
`template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;`

```
template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
```
[24, §20.8.2.2.1] which should be eligible for overload resolution if and only if `T*` is convertible to `Y*`.

This entails that a `shared_ptr<A>` shall be implicit convertible to `shared_ptr<B>` if `A` is a derived class from `B` and `B` is accessible and unambiguous. But this conversion is costly, because of the reference counting, which in this case decrements the reference count in `shared_ptr<Derived>` and increments in `shared_ptr<Base>`. This increment, decrementing is unnecessary and resource consuming.

To solve this problem, the easiest method is to refactor the function which expects a `shared_ptr<Base>`, to expect a `const Base&` or `const B*` and call the function with either `*ptr.get()` or `ptr.get()`.

4.1.2   Inefficient Stream Use Check

This problem consists of two parts. Firstly what makes it marginally inefficient to stream characters as strings. The standard defines that the hierarchy of the classes which handle Input/Output should look like in Figure:10.

Figure 10: C++ IO hierarchy
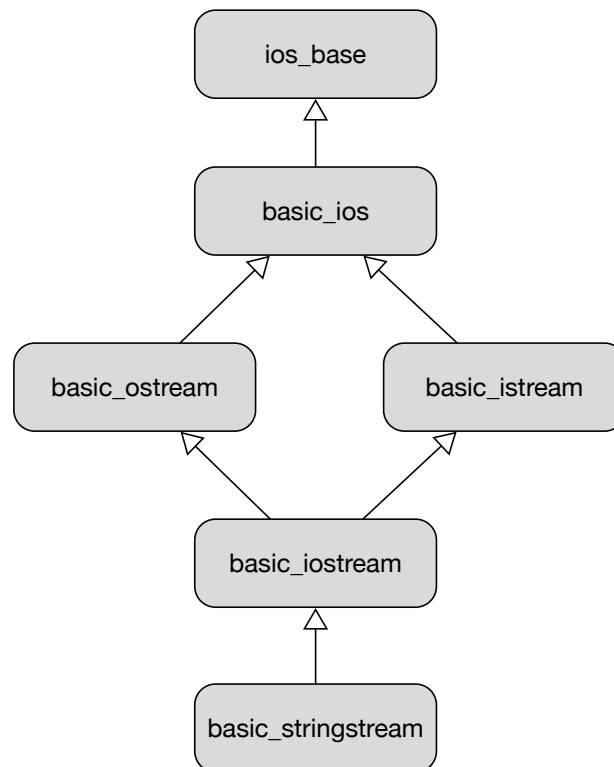
In `basic_ostream` there are several overloads of `operator<<` however we are only interested in two particular overload:

- ```
  template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
  char);
  ```

- ```
  template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
  const char*);
  ```

[24, §27.7.3.6.4] These overloads enable the `basic_ostream` object to accept both `<< "A"` and `<< 'A'` structures.

These two overloads both achieve the same goal, however performance-wise they are not the same, since `"A"` structure will be represented by `const char[2]`[1] in the code while `'A'` will be only a single instance `char`. And since arrays can be implicitly converted to pointers[24, §4.2] the former version will call the overload with `const char*` in its signature.

The only problem with `const char*` is that the length of the string is not known, so it must be processed until the tailing null-byte is found. On the contrary, when using `char` the length is known, so no extra computation is required. However since modern CPUs are quite fast this extra computational complexity barely affects the observable performance, but in an embedded or in other time-critical even this small performance gain is significant.

The other inefficiency related to `basic_ostream` are the multiple calls to `endl` function which is defined as[24, §27.7.3.8]:

```
template <class charT, class traits>
  basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
```

To be able to apply `operator<<` on functions the standard defines the following[24, §27.7.3.6]:

```
basic_ostream<charT,traits>& operator<<(
    basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
```

which enables `operator<<` to take a function, that returns `basic_ostream&`, as an argument.

With this information one could think that streaming more than one `std::endl` is equivalent to streaming `'\n'` characters. However the standard also states that the `endl` function has to call `flush`[24, §27.7.3.8] which flushes the output buffer,

---

[1]strings are closed by null bytes in C++ so it would look like 'A'\0

which is a time consuming procedure. The actual performance overhead caused by flushing the output buffer is operating system dependent, but in general it can be categorised as a performance hog, if used repeatedly.

4.1.3 Inefficient String Concatenation Check

As the C++ standard defines the Strings library it provides several ways of concatenating two strings together, `a = a + b;`, `a += b;` and `a.append(b);`.

The first one requires two operators to be declared, the first is `operator+` for `basic_string` [24, §21.4.8.1]:

```
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator>
  operator+(const basic_string<charT,traits,Allocator>& lhs,
            const basic_string<charT,traits,Allocator>& rhs);
```

And also the `basic_string`'s relevant constructor:

```
basic_string(basic_string&& str) noexcept;
```

The other two ways of concatenating is with a designated member function: [24, §21.4.6.1] and [24, §21.4.6.2]

```
basic_string&  operator+=(const basic_string& str);
basic_string&  append(const basic_string& str);
```

Where `operator+=` just calls `append`. `append` then does the process of concatenating the strings. The standard does not say how the implementation should behave in the `append` operation, but an industrial grade standard implementation should use the possibility of simply copying (moving if applicable) the characters of the `append` argument to the receiver's buffer if it fits. Otherwise the buffer should be increased and the whole temporary object copied.

This way it would be more efficient to use `append` member function instead of creating a new string with containing the exact copy of itself and then assigning it to itself.

4.1.4 Default Container Initialization Check

The C++ standard defines standard containers which are able to hold collections of objects. The containers which are defined and we are interested in are: `std::vector`, `std::set`, `std::deque` and `std::map`. To put elements in these containers one can

define them with the default constructor and later add elements one by one with insertion operations.

Each insertion is a call of a member method for instance `push_back()`, `insert()` and `emplace_back()`. To use these methods right after creating the container is inefficient because the standard mentions that every mentioned container must have a constructor which takes an initializer list as an argument:

- `vector(initializer_list<T>, const Allocator& = Allocator());` where `T` is the Type of the objects in the container [24, §23.3.6.2]

- `map(initializer_list<value_type>...);` where `value_type` is a pair of the key and value type of the `map` [24, §23.4.4.2]

- `set(initializer_list<value_type>...);` where `value_type` is the type of the `set`'s key [24, §23.4.6.2]

- `deque(initializer_list<T>, const Allocator& = Allocator());` where `T` is the Type of the objects in the container[24, §23.3.3.2]

The more efficient form would be list-initialising the containers in place with the declaration by passing an `initializer_list` to the constructor. But the list-initialisation raises concerns when implicit casting is involved. The standard states "If a narrowing conversion . . . is required to convert any of the arguments, the program is ill-formed"[24, §8.5.4 3.6].

A narrowing conversion is an implicit conversion of numerical types where there is a possibility of over- or underflow when converting to the destination type. For a more detailed description see [24, §8.5.4 7]. To remove implicit narrowing conversions we need to introduce explicit casting. This checker uses C-style casts, which in the future should be replaced with a safer `static_cast`.

The standard also ensures that the elements in a list-initializer are constructed from left to right order[24, §8.5.4.4], this way not violating the order they were originally added to the container.

4.2   Implementations

In this section I am going to cover the implementation details of each module, covering the possible pitfalls and issues with implementing a standalone Clang-tidy module.

4.2.1   General implementation details

Each check have at least two methods, which are inherited from the `ClangTidyCheck` base class. The first method is `registerMatchers` which acts as a set-up phase. The actual AST matcher structure should be assembled in this function, adding all components. Also, the callback method is specified in this segment, more precisely the object on which the callback method is called, this is usually `this`. In this method, the developer has access to a `MatchFinder` object which controls the matchers. To add an AST matcher to the `MatchFinder`, the member function `addMatcher()` can be used, that takes an AST matcher and a callback object as parameters.

The other method is the callback function named `check`. This method gets called for every match on the AST by the checkers registered beforehand in the `registerMatchers` method. In this function, the developer has access to virtually everything from the actual source code to the AST nodes, which can be queried from a `MatchResult` object.

4.2.2   Inefficient String Concatenation Check

Since Clang 4.0.0[25] this module is part of the official Clang svn repository and had been referenced in a guide to improve developer workflow productivity[26].

When implementing this checker the greatest challenge was matching such `operator=` operator calls which have the left-hand side variable on the right-hand side also. Initially, there were two options, either filter on the AST matcher's side or match everything and then filter in the callback function.

Filtering in the AST matcher is the preferred method because it's less resource intensive and clearer. This is handled by first searching for an `operator=` with a `basic_string` on the left side and binding a label, basically a string, to that node. Then traversing the right side of the operator and looking for `operator+` and also the node which is equal to the already bound node.

Since the checker can be parameterized there is a third method overload named `storeOptions` which handles the passed parameters. If the option `StrictMode` is specified as `true` the checker will be triggered for all inefficient calls, else only for those which are inside a loop.

This way we are sure that the callback will only be called for those operators which have the same variable on the right side as the left side.

4.2.3   Inefficient Stream Use Check

The implementation fo this checker consists of two distinct parts, one part for matching the character conversions and the other is for matching the multiple uses of `std::endl`.

To match the character conversion, the tool is looking for calls to `operator<<` that have `std::basic_ostream` as one of the arguments, to make sure we are only dealing with cases regarding streams and not interfering with user-defined overloads of `operator<<`. The other argument must be an `ImplicitCastExpr`, that as the name suggests is an implicit cast between two types, which have `const char[2]` as source and `const char *` as a destination. With these structures, we can safely match the inefficient character conversions.

To match multiple `std::endl` use, we can use the left-associative property of `operator<<`. Matching the outermost `std::endl` and looking for any descendant `std::endl` we can make sure only the inner `std::endl` calls will be labelled.

Having constructed the checking functionality so that it only calls the callback function on an exact match we have only formatting to do. In case of `std::endl` the formatting is quite simple, just replace the unwanted `std::endl`s with '\\n' to be able to print '\n\ in the warning message.

In the case of simple characters its a more complex process, because of the character escapes. This is solved by introducing a static free function `getEscapedChar` which contains a character table and maps the characters to their escaped versions.

4.2.4   Default Container Initialization Check

This module is the most complex in terms of implementation details and structural complexity. To match the smallest scope of code in the `registerMatchers` function we have to introduce several variables. Beforehand as a high-level overview of the AST matcher, we want to define. It should match on such `CompoundStmt`s that contain a `CXXMemberCallExpr`, but are not inside of a template instantiation.

First of all, we define the declarations of the insertion methods (`insert, push_back, emplace, emplace_back`), and also the containers we are interested in (`map, vector, set, deque`). Also, we need a matcher that matches `CXXMemberCallExpr`s which reference the container and have the desired insertion declaration.

We need three types of `CXXMemberCallExpr` to match all the possible scenarios. First, we define a variable named `MemberCallExpr` to match `CXXMemberCallExpr` on

the previously defined containers, with the specific insertion type. Second we define a variable named `MemberCallExprWithRefToContainer` which is an extension of `MemberExpr` having a reference to the original container as argument, for instance this would match `vec.push_back(vec.size())`.

The third helper variable is `UnresolvedMemberExpr` which matches `UnresolvedMemberExpr`. To match this expressions a custom AST matcher had to be created which is specialised on `UnresolvedMemberExpr` and also a matcher that can check the base of such an expression.

The first checker to be added is matching a `CompoundStmt` which is not inside of a template instantiation and matches for each `UnresolvedMemberExpr`, the second matcher is the same as the first matcher but matching a `MemberCallExprWithRefToContainer` inside a `CompoundStmt`. The third checker is simply matching `MemberExpr`s inside a `CompoundStmt`. With these three matchers we can pass the callback function optionally matched AST nodes indicating where to stop processing the nodes.

In the callback function we are walking the direct children of the matched `CompoundStmt` til we find the container. We process step by step all the `CXXMemberCallExpr`s following the container declaration until there's no more expressions or either the matched `UnresolvedMemberExpr` or `MemberCallExprWithRefToContainer` nodes.

For every `CXXMemberCallExpr` we extract the arguments with `getInsertionArguments` and format them with `formatArguments`, adding all the required narrowing conversion by calling `getNarrowingCastStr`. After having all the arguments formatted, we create a removal of the insertion expressions with `getRangeWithFollowingToken` which calls the parser to get the next token after the expression. After emitting the diagnostic messages, we add the container to the already processes container declarations.

4.2.5   Shared Pointer Conversion Check

To summarise in a nutshell the matcher part of the check, it's looking for `CallExpr`s involving such functions that have a `shared_ptr` as an argument and a constructor-conversion is taking place at the place of the function call.

Delving more into the details of the matcher implementation, I should highlight the structure responsible for matching the actual constructor-conversion. To do that we need to define two `QualType`s to represent the type for `shared_ptr` and also `unique_ptr` (I will talk about its importance later) with the help of the function

`getQualTypeForTemplate`, which is a C++11 function using tailing return type with `auto` type deduction of `decltype(QualType())`.

Having the aforementioned types, we can construct the part which matches on such `Exprs` which have an `ImplicitCastExpr` that is a `CK_ConstructorConversion` type of cast. Moreover, this cast has to have the type of the previously mentioned `shared_ptr` unless it is a conversion from `unique_ptr` to `shared_ptr`.

With these components, we only need to match such `CallExprs` that have this implicit conversion and bind a label to it, and also bind a label to the calllee function's declaration.

In the callback function our only task is to resolve the base and the derived type to be able to print a meaningful diagnostic message involving the type names. Getting the derived class's name is relatively easy since the standard specifies[24, §20.8.2.2.1] that the first template argument of the converting constructor must be the type it's converting from. So we only need to query the first template argument's declared name.

Resolving the name of the base class is not quite this straightforward because the type information is not encoded in such an explicit way in the expressions we already matched. The information about the base type is encoded in the destination type of the implicit cast, which can be retrieved by using the `getBaseTypeAsString` static free function.

Having both types resolved we can print diagnostics as a warning on the actual function call and as a note pointing at the location of the parameter of the callee function, that it should be rewritten to a more efficient code.

4.3   Program Structure

This section covers the abstract structure of the application in the Clang-tidy project. The visualisation is done with the Unified Modelling Language (UML for short), which is a standard way of representing different logical and physical connections in a computer software. We use UML Class diagrams to visualise the methods and fields of an object, representing also the visibility of these components from the outside of the class.

In Figure:11 we visualise each module's ancestor and how these modules are decoupled from each other.

Figure 11: Inheritance diagram
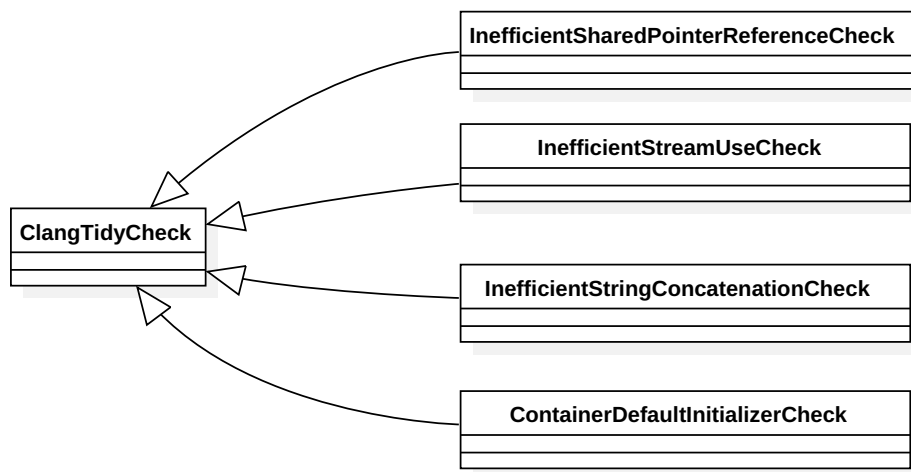
Every module extends from the `ClangTidyCheck` class, which provides the interface of a Clang-tidy module to create the AST-matchers and also how the matched code should be handled afterwards. The first step of adding a new module to Clang-tidy is to inherit from this class and implement the methods, which can be used also to send extra parameters to the checkers themselves.

Figure 12: UML class diagram of Inefficient Stream Use checker
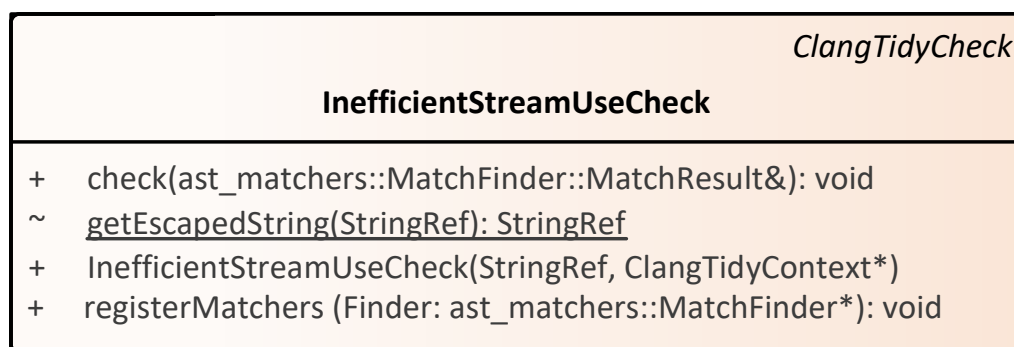
In Figure:12 you can observe one of the simplest module's UML Class diagram, which implements the methods declared by `ClangTidyCheck` and also defines one static free function in the source file.

Figure 13: UML class diagram of Container Default Initializer check
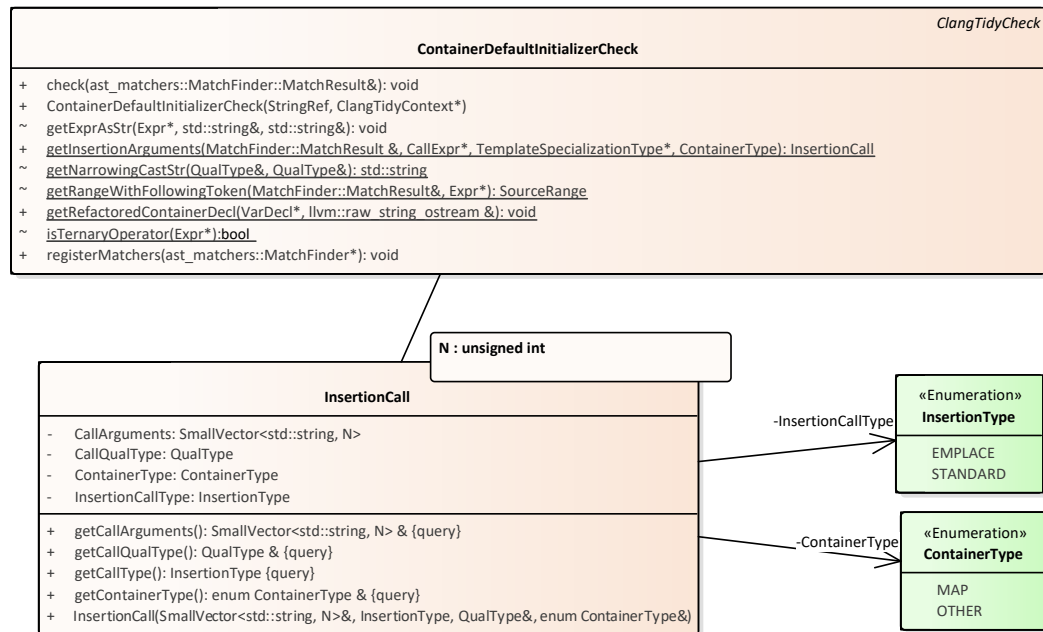
Figure:13 represents the abstract structure of the Container Default Initializer check. The solved problem in this module is more complex than the rest hence the increased complexity in the abstract structure. This checker uses several helper functions, and utility classes to store all the meta information about the containers and the conversions taking place in instantiating them.

Figure 14: UML class diagram of Inefficient String Concatenation check
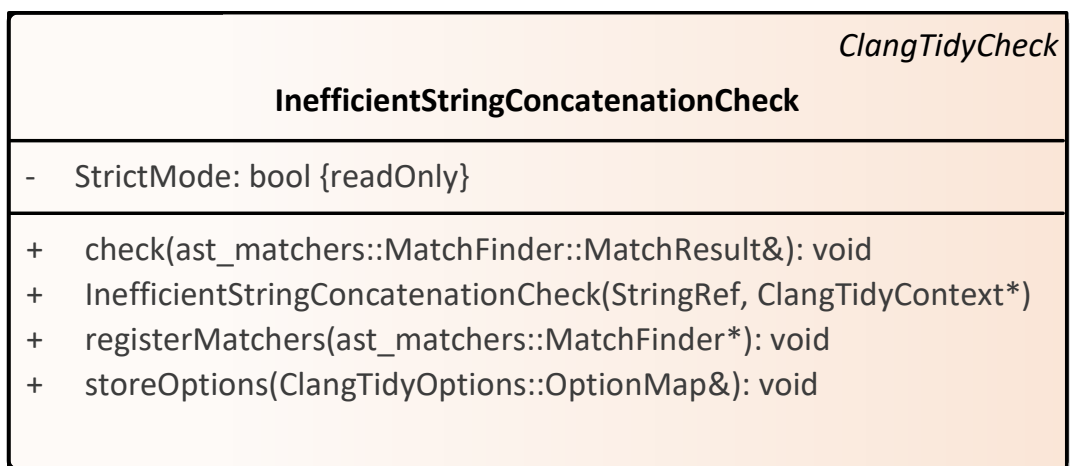
The next module's architecture is fairly simple again, as seen in Figure:14. It

consists of the basic interface of a Clang-tidy checker, but it uses a new method `storeOptions` to be able to receive a passed parameter and modify its behaviour accordingly.

Figure 15: UML class diagram of Inefficient Shared Pointer check

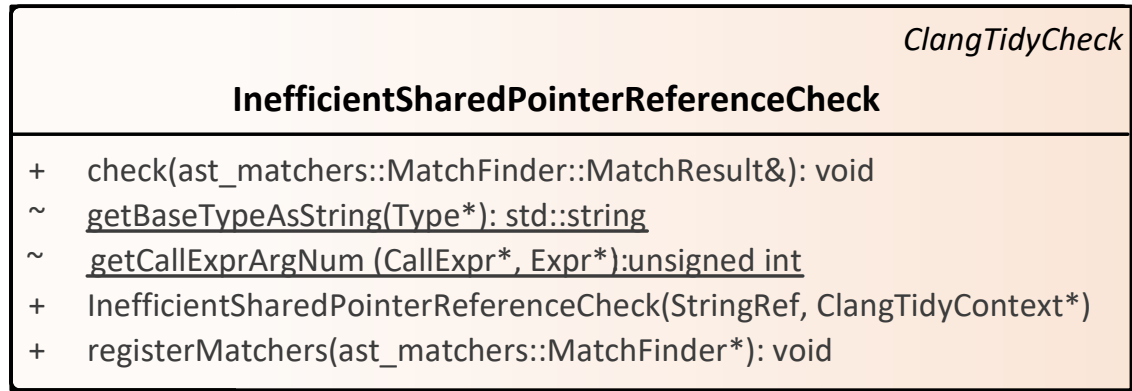| *ClangTidyCheck* |
|---|
| **InefficientSharedPointerReferenceCheck** |
| +   check(ast_matchers::MatchFinder::MatchResult&): void |
| ~   <u>getBaseTypeAsString(Type*): std::string</u> |
| ~   <u>getCallExprArgNum (CallExpr*, Expr*):unsigned int</u> |
| +   InefficientSharedPointerReferenceCheck(StringRef, ClangTidyContext*) |
| +   registerMatchers(ast_matchers::MatchFinder*): void |

In Figure:15 you can inspect the structure of the Inefficient Shared Pointer checker which declares two extra static methods for retrieving extra information about the matched nodes.

5 Testing

Testing is one of the most important parts of developing new software. Without the required quality assurance measures, we cannot produce justifiably safe code. Clang-tidy offers an out of the box solution for writing tests for the new modules, which can be run as a separate build task, to support the continuous integration of the project.

5.1 Tools for testing

5.1.1 Python

Python is a multi-paradigm dynamically typed programming language. It can be interpreted line by line and also it has a huge amount of libraries which can extend the language. It's one of the most popular programming languages in the world. It emphasises code simplicity and in general, requires fewer lines of code to achieve the same result as with C++ for example.

The aforementioned characteristics of the language makes it perfectly suitable to be a C++ testing suite. Clang-tidy uses numerous of Python scripts, from making it easier to add new checkers to the project to running a custom configured Clang-tidy to all the files in a project.

5.1.2 CMake

As I have already mentioned previously CMake is a framework for both building and testing applications. In LLVM and its subprojects define several CMake tasks which will run the tests on the project and verify their result and fail the task if one of the tasks failed.

5.1.3 FileCheck

FileCheck is a utility script in Clang project which simplifies the testing of tools that need to emit some sort of an output[27]. In Clang-tidy this tool is used to verify that the tool, run on a test-file, produces the expected output, as well as the expected FixItHints were applied to the file.

Along with `check_clang_tidy.py` which is a driver for FileCheck, their responsibility is to ensure that every test-file is executed and all the expectations are met. The `check_clang_tidy.py` is executed as part of the `check-clang-tools` CMake process.

The assertions are annotated as C++ comments, with prefixes that are Clang-tidy specific (`CHECK-FIXES, CHECK-MESSAGES`). To verify an output message, a diagnostic output for instance, one can use the `CHECK-MESSAGES` command, which must be parameterized with line and row numbers of the exact position of the expected message, for instance `CHECK-MESSAGES: :[[@LINE-1]]:2: warning: message` means that we are expecting "warning: message" diagnostic message to be emitted at the second column at the line before the current line. This way we can pinpoint the location of the emitted messages.

FileCheck also supports regular expressions inside the validation code inside `{{ }}`. If the code to match contains `{{` one can use `{{[{][{]}}` to escape the bracket characters. Using regular expressions is especially useful when validating applied FixItHints for making the expected string more concise and ignore possible platform dependent whitespace issues.

5.2   Regression tests

The structural stability and validity of the implemented modules were tested with the help of regression tests. These consist of single files that contain several code snippets with the problematic code patterns present and also in absence of such code to minimise accidental false-positive fixes or warning messages.

Testing and validating Clang-tidy modules can be a challenging process and has to be taken with care to decrease the number of possible false positive matches or wrong FixItHints which can break the further compilation of the file or whole project.

5.2.1   Testing templated code

The problem with C++ templates is that the compiler creates distinct functions or classes depending on the template parameters before the Abstract Syntax Tree is built. The result is having the same problematic code pattern multiple times in the AST, and having each templated instantiation with the same location information as the original one.

This means that if we naively match the AST not checking whether it's a template instantiation that we are matching, we can correct the code multiple times each pointing to the original template declaration. This can result in nondeterministic FixItHints which point to the same source location with different content.

Outlining such a scenario as a motivating example:

```
template<int N>
struct A{
    int f(){ return N; }
};

template<int N>
void f1(A<N>& a) {
    std::vector<int> vec;
    vec.push_back(a.f());
    vec.push_back(N);
}

int main() {
    auto a1 = A<4>{};
    auto a2 = A<5>{};
    f1(a1);
    f1(a2);
}
```

Given the above code and a checker, similar to Container Default Initializer checker, which looks for consecutive `CXXMemberCallExpr` objects after the default constructor, gathers them and fixes the container initialization. This checker would find that the expression at `line 9` is a `CallExpr` wrapping an `UnresolvedMemberCallExpr` which is right because that expression is dependent upon the template parameter `N`.

With this information the checker does nothing but skip this initialisation and moving to the next one. Because in this example there are 3 initialisations of `std::vector`s. The remaining two comes from the template instantiation of `f1` by calling it with both `a1` and `a2`.

When this checker is at the second instantiation which is with `N = 4`, there is no dependent expression anymore, so it creates a FixItHint `vec{a.f(), 4}` to replace the default constructor with. After replacement, the checker creates a FixItHint again, with `vec{a.f(), 5}` since `N = 5` in that instantiation.

So in the end we have two distinct FixItHints `vec{a.f(), 4}` and `vec{a.f(), 5}`, having the same source location to apply them. But how to decide which one to apply, or should we apply them at all? The answer is that we should not apply either of them because they are all just a product of template instantiation.

To solve these type of problems which are rooted in templated code instantiation we should check whether the AST segment we are matching against is a template

instantiation.

5.2.2 Mocking

To be able to test the modules' behaviour independent of the actual standard implementation when using elements from the standard library, we have to create an isolated environment imitating the way how the standard works. To achieve this when using the STL library we create the required classes and functionality. Objects created to mimic the behaviour of the original one are called mock objects.

5.2.3 Test files

There is a test-file for each module introduced, which are run as part of `check-clang-tools` CMake task. Each file contains the mock of the related STL containers and other functionality residing in namespace `std`, for example, conversions, templated type checks, etc..

Each test-case in the files are annotated with `CHECK-MESSAGES` and/or `CHECK-FIXES` comments. After them, we formulate the expected diagnostic messages and also the expected FixItHints fixes.

6   Summary

To summarise, with the use of static program analysis the four new modules introduced in this document, that are part of the open-source project Clang-tidy, are able to find code patterns, in C++ code-base, which are inefficient constructs in terms of runtime performance. Using source-to-source transformations two modules have the ability to automatically correct the inefficient part of the source code.

The four modules are able to warn about the concatenation of `std::string`s in a more efficient way, to use `operator<<` without performance overhead, initializing `STL` containers in place and also to use `std::shared_ptr` with caution.

Having run several benchmarks on the problematic code snippets, significant performance differences begin to emerge between the refactored and the original code. This justifies the goal of this thesis in creating a usable code-quality improving software.

7 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] "Cmake," 2017. `https://cmake.org/`.

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

[11] "Mingw - minimalist gnu for windows," 2017. `http://www.mingw.org/`.

[12] "Msys," 2017. `http://www.mingw.org/wiki/MSYS`.

[13] "Windows subsystem for linux overview," 2017. `https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/`.

[14] "Drawbridge," 2017. `https://www.microsoft.com/en-us/research/project/drawbridge/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fdrawbridge%2F`.

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

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

[17] "Json compilation database format specification," 2017. `https://clang.llvm.org/docs/JSONCompilationDatabase.html`.

[18] "The ninja build system," 2017. `https://ninja-build.org/manual.html`.

[19] "Clang 4.0.0 release notes," 2017. `http://releases.llvm.org/4.0.0/tools/clang/docs/ReleaseNotes.html`.

[20] "Clion starts 2017.2 eap with clang-tidy integration," April 20, 2017. `https://blog.jetbrains.com/clion/2017/04/clion-2017-2-eap-clang-tidy/l`.

[21] "Clion clang-tidy integration," 02.04.2017. `https://plugins.jetbrains.com/plugin/8301-clion-clang-tidy-integration`.

[22] "Research projects," 2017.05.05. `http://gsd.web.elte.hu/projects/`.

[23] "Codechecker," 2017.05.05. `https://github.com/Ericsson/codechecker`.

[24] "Working draft, standard for programming language c++," 2015-11-09. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4567.pdf`.

[25] "Extra clang tools 4.0.0 release notes," 2017. `http://releases.llvm.org/4.0.0/tools/clang/tools/extra/docs/ReleaseNotes.html`.

[26] "Improving workflow by using clang-based tools," Aug 30, 2016. `https://omtcyfz.github.io/2016/08/30/Improving-workflow-by-using-Clang-based-tools.html`.

[27] "Filecheck - flexible pattern matching file verifier," 2017. `http://llvm.org/docs/CommandGuide/FileCheck.html`.