TU WIEN Informatics

Erkennen von Leistungsengpässen mittels AST-Transformationen

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Maximilian Alfred Hagn
Matrikelnummer 11808237

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Wien, 15. Jänner 2022

Maximilian Alfred Hagn	Sascha Hunold

TU WIEN Informatics

Detecting Performance Bottlenecks through AST Transformation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Business Informatics

by

Maximilian Alfred Hagn
Registration Number 11808237

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Vienna, 15th January, 2022

Maximilian Alfred Hagn	Sascha Hunold

Erklärung zur Verfassung der Arbeit

Maximilian Alfred Hagn

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Jänner 2022

Maximilian Alfred Hagn

Kurzfassung

Bei der Entwicklung von Software stehen viele Programmiererinnen und Programmierer früher oder später vor der Frage, wie sie ihre Anwendungen optimieren und damit auch die Laufzeit so kurz wie möglich halten können. Gerade im Bereich des Hochleistungsrechnens und bei der Entwicklung zeitkritischer Software ist die Verkürzung der Laufzeit von Programmen unabdingbar. Die Laufzeitanalyse ist daher eine grundlegende Technik, die jede Entwicklerin und jeder Entwickler beherrschen muss. Zu diesem Zweck wird der Quellcode entweder manuell überprüft oder die Anwendung während der Laufzeit mit Profiling-Tools analysiert. Diese Instrumente bieten den Benutzenden die Möglichkeit, Daten über die Leistung einzelner Funktionen zu erhalten. Bei den meisten Tools können die *interessierenden Bereiche(ROI)* in der Regel manuell festgelegt werden. Die Identifizierung dieser Bereiche kann problematisch sein, insbesondere bei der Entwicklung komplexer Programme. Daher wäre es wünschenswert, wenn es eine Methode gäbe, mit der benutzerdefinierte *Bereiche von Interesse* durch automatisches Einfügen von Messcode analysiert werden könnten.

In dieser Arbeit wird ein Werkzeug zur Leistungsmessung vorgestellt, das zur automatischen Messung der Laufzeiten kritischer Bereiche in *C/C++* Anwendungen verwendet werden kann. Das entwickelte Werkzeug basiert auf dem von *Clang* bereitgestellten Framework, das verschiedene Funktionen zum Traversieren von *abstrakten Syntaxbäumen(AST)* bietet. Das Werkzeug analysiert den von *Clang* bereitgestellten *Syntaxbaum* des Eingabecodes und fügt automatisch Messcode an den gewünschten Stellen ein. Dazu wird der *abstrakte Syntaxbaum* von der Wurzel abwärts durchlaufen, wobei in jeder Iteration eine hierarchisch niedrigere Ebene von der Benutzerin oder dem Benutzer ausgewählt werden kann. In jeder Iteration erhält die oder der Anwendende Statistiken über die *interessierenden Bereiche* und kann selbst auf Basis der gewonnenen Informationen entscheiden, welcher Bereich als nächster analysiert werden soll. Um die Funktionalität des Programms vorzustellen werden wir den Arbeitsablauf an verschiedenen Beispielen demonstrieren. Des Weiteren wird anhand einer Überlaufanalyse gezeigt, dass der Überlauf des Profiling-Prozesses durch die iterative Vorgehensweise gering gehalten werden kann.

Abstract

When developing software, many programmers are sooner or later faced with the question of how they can optimize their applications and also keep the runtime as short as possible. Especially in the field of high-performance computing and in the development of time-critical software, shortening the runtime of programs is indispensable. Runtime analysis is therefore a fundamental technique that must be mastered by every developer. For this purpose, the source code is either checked manually or the application is analyzed during runtime using profiling tools. These instruments offer the possibility to obtain data about the performance of single functions for the user. In most tools, *Regions of Interest (ROI)* can usually be specified manually. Identifying these can be problematic, especially when developing complex programs. Thus, it would be preferable if a method existed with which user-defined *regions of interest* could be analyzed by automatically inserting measurement code.

This thesis presents a performance measurement tool that can be used to automatically measure the runtimes of critical areas in *C/C++* applications. The developed tool is based on the compiler front-end *Clang*, which offers various functions for traversing *Abstract Syntax Trees (AST)*. The tool analyzes the *abstract syntax tree* of the input code provided by *Clang* and automatically inserts measurement code at the desired positions. For this purpose, the *abstract syntax tree* is traversed from the root downwards and in each iteration a hierarchically lower level can be selected by the user. In each iteration, the user receives statistics about the *regions of interest* and can decide which area to analyze next based on the information obtained. To demonstrate the functionality of the program we will show the workflow on different examples. Furthermore, an overflow analysis will be carried out to show that the overflow of the profiling process can be minimized by using an iterative approach.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Performance analysis is one of the most important tools for developing high-quality software applications. For this purpose, the developed program code is either reviewed manually or the runtime of individual program sections is measured using performance counters within the code. This method is particularly essential in the field of high-performance computing and in the development of time-critical programs. However, performance and runtime can also be used to measure quality of all other applications. In *OpenMP* and *MPI* applications, the techniques profiling and tracing are used for performance analysis and consequently for performance improvement. As a result of the tracing method, all events that occur are visualized in a diagram, which enables a comprehensive analysis of the program code. However, the lightweight method of profiling also enables the user to find performance bottlenecks. This is usually accomplished by wrapping functions with performance counters and finally measuring the elapsed time within the functions at runtime. Thus now, the user can find out how often a function has been called and how much time each call takes. If more detailed information is needed, in most conventional profiling tools the user can determine code *Regions of Interest (ROI)* that are to be analyzed in more detail. For this, however, the user must have full knowledge of the underlying program code. Furthermore, the regions that are considered critical must be known before further analysis can be carried out. In order to address these problems, this thesis aims to develop an application that displays critical areas and allows immediate closer analysis on user-defined *regions of interest*.

1.1 Motivation

We have already explained that most conventional profiling tools work at the function level. This means that only larger sections are analyzed at a time, which results in information about the runtime of individual *statements* and *declarations* remaining hidden.

Listing 1.1: Example Code Showing a Simple *C++* Application.

```
void loopFunc() {
    for { /* code loop */ }
    for { /* code loop */ }
    for { /* code loop */ }
}

int main(void) {
    loopFunc();
    return 0;
}
```

Listing 1.2: The Compilation and Execution of the *gprof* Profiler.

```
gcc -pg -lstdc++ app.cpp -o app     // compilation of the program
./app                               // execution of the program
gprof app gmon.out                  // start runtime analysis

/* output generated by gprof */
Flat profile

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
100.00      0.01     0.01    50000    20.00    20.00  loopFunc()
  0.00      0.01     0.00        1     0.00     0.00  overhead
```

Listing 1.1 shows a simple *C++* application in which a function `loopFunc()` containing three `for`-loops is called.

To analyze this program with the common profiler *gprof* [1], we can compile the code with the help of *GCC*, a *Compiler Collection* for various languages including *C* and *C++* [2], as shown at the start of Listing 1.2. Afterwards, the application has to be executed so that analysis data can be collected. Finally, we can use *gprof* to analyze the program [3]. It can be seen that the profiler returns runtime statistics for the `loopFunc()` function, but misses detailed data about the separate `for`-loops. Even if the user could find performance bottlenecks on the function level, there is no information about the individual loops or *declaration statements*. However, the user could define specific code *regions of interest* to particularize the output prior to the execution of the application [4]. To improve the profiling process for the user, we want to instrument the *regions of interest* and implement them automatically. Therefore, the user could find performance bottlenecks not only on the function level, but also for individual *statements* or *declarations*.

Listing 1.3: Example Code Showing the Instructions Wrapped by Performance Counters.

```
void loopFunc() {
    startEvent(1);              // inserted automatically
    for { /* code loop */ }
    endEvent(1);                // inserted automatically
    startEvent(2);              // inserted automatically
    for { /* code loop */ }
    endEvent(2);                // inserted automatically
    startEvent(3);              // inserted automatically
    for { /* code loop */ }
    endEvent(3);                // inserted automatically
}

int main(void) {
    startEvent(0);              // inserted automatically
    loopFunc();
    endEvent(0);                // inserted automatically
    printCollectedData();       // inserted automatically
    return 0;
}
```

1.2 Goal of the Thesis

The aim of this thesis is to analyze bottlenecks in *C++* source codes in a programmatic way and therefore insert an individually adaptable performance counter automatically. For instance, we could assume that our application looks similar to the one in Listing 1.1. An application based on the *Clang* [5] *Abstract Syntax Tree (AST)* transformation framework is to be developed that can analyze and transform the source code. The program should take a single-threaded *C++* application as input and finally return a program that contains performance counters wrapped around the *regions of interest*. As shown in Listing 1.3, we can use our transformation tool not only to insert specific events at the function level, but also before and after specific instructions. From this point on, we will refer to the developed tool as *ROIProfiler*, which derives from *Regions of Interest Profiler*. The developed tool is freely available in a GitHub repository [6].

Another specification of the *ROIProfiler* is that the transformation should be done hierarchically, which means that in the first transformation step only the outer scopes are wrapped, and in subsequent steps, also inner scopes are wrapped. Thus, the first time the *ROIProfiler* is executed, only the top level instructions should be measured in order to give the user first reference points for a performance analysis. The user receives a detailed list of the resources used within the program, with instructions returned as pairs with identification numbers. These statistics are shown in Table 1.1, where it can be seen that in addition to the runtime of individual sections, information is also given about the proportion to the total resources used. Furthermore, the statistics also provide data about the resources used by the inserted performance counters.

Table 1.1: The Desired Output of the Transformed Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
2192956	ForStmt	402.291 µs	4.62 %	4.62 %	1
2193096	ForStmt	2.671 µs	0.03 %	0.03 %	1
2225085	ForStmt	8303.364 µs	95.34 %	95.34 %	1
Overhead		0.626 µs	< 0.01 %	< 0.01 %	12
Runtime		8709.478 µs			

If the user wants to obtain further statistics about a specific region, the *ROIProfiler* can be executed again, this time simply providing the identification numbers from the statistics. The method presented makes it possible to profile an application level by level. The generated statistics provide the user with an overview of particularly resource-intensive regions, which can be examined more closely through the incremental execution of the *ROIProfiler*. Thus, the *ROIProfiler* makes it possible to identify performance bottlenecks in software applications in a programmatic way. In order to prove that the developed *ROIProfiler* meets the specifications, a detailed functionality test and an overhead analysis are carried out. To check the desired functionality, applications of different complexity levels are profiled with the developed approach. In order to determine the overhead generated by the *ROIProfiler*, we run a set of programs before and after adding performance counters to check how the additional resource consumption influences the runtimes.

1.3 Structure of the Work

The thesis starts with a focus on the observation of *Clang* as the underlying framework in Chapter 2. Furthermore, the basic functionality of *LLVM* as the back-end part of a compiler and *Clang* as the front-end framework is explained. To round off Chapter 2, we take a closer look at the representation used by *Clang*, as a fundamental understanding of it is essential for the development of our software. Chapter 3 describes how the program code can be analyzed to find the desired locations in the application where the performance counters are to be positioned. The approach used to insert it dynamically and the possibilities that arise from this technique are described in Chapter 4. Furthermore, the runtime calculation and the output of the statistics are explained in this chapter. In Chapter 5 we focus on the results of the research and further examine the performance of the developed tool more precisely. In order to enable an accurate evaluation, both a functionality test and an overhead analysis are carried out. In Chapter 6 we address the limitations of the developed approach, especially by considering the possibilities for overhead reduction and the use for multi-threaded applications. Finally, we summarize all the findings of the research and clarify the need for further research and possible improvements.

CHAPTER 2

Software Framework for the Transformation Tool

Automatic analysis and modification of program code is a widely applied technique to quickly optimize and customize many complex programs at once. *Clang* provides a library that makes it possible to write tools that can be used primarily for *source-to-source* transformation of *C*, *C++*, and *Objective-C* programs [7]. This interface enables the developer to use the powerful *abstract syntax tree* and the extensively developed infrastructure of *Clang*. For example, Fabian Schlebush et al. [8] used the infrastructure to categorize high performance computation applications. Another example is the approach of Alexander Hück et al. [9] to rewrite incompatible code resulting from upgrading to newer versions into error free code. In order to develop a tool that is suitable for wrapping *regions of interest* with performance counters, we will also use the *LLVM* compiler infrastructure and the front-end framework *Clang* that is built on top of it. The former is a modern compiler architecture that contains a variety of tools and libraries to optimize high-level languages, like *C++*, during translation into machine code [10]. *Clang* allows to analyze, structure, and check the code for semantic and syntactic errors [5]. Furthermore, the front-end part of the compiler is capable of transforming the program code into an *abstract syntax tree*, which can then be translated into machine language by the back-end. Especially the front-end part is interesting for our type of application, as the transformation from *source-to-source* is supported by predefined macro functions that combine a large number of individual instructions. This chapter explains the fundamental concepts behind the software framework and focuses on the *abstract syntax tree* of *C++* applications created by *Clang*.

2.1 The LLVM Compiler Infrastructure

The back-end part of the compiler is responsible for transforming the provided syntax tree into machine code. To achieve this, *LLVM* creates the so-called *LLVM intermediate representation*. This is an intermediate machine code that can be used to abstract different programming languages. This code can now be used to apply optimization methods independently of the input language. After all optimizations have been completed, the intermediate code can be compiled into machine code, again with the possibility of addressing different processor architectures and thus being target-independent [11]. However, as the *LLVM* project has grown, several sub-projects have been added, one of which is the *Clang* front-end framework. This project deals with the fast and efficient compilation of *C*, *C++*, and *Objective-C* applications [12].

2.2 Clang Language Frontend for LLVM

Clang is a front-end framework for *LLVM* that is capable of translating programming languages from the *C* family and also provides an infrastructure for the development of tools [5]. As a front-end part of a compiler architecture, one of the most fundamental tasks is the creation of an *abstract syntax tree*. In general, an *abstract syntax tree* is a generalized representation of the written programming language, where a node represents an instruction. To create such a model, the given source code must be analyzed and checked for errors so that it can finally be represented in a structured way.

In *Clang*, these nodes do not contain any information about dependencies, which makes the system very optimized in terms of memory consumption and speed. The data about the connections between the nodes and the general structure of the program is stored in the so-called *identifier table* [13, 14]. In order for the user to navigate through the model, an *entry point* is needed, which in *Clang* is called `ASTContext` and can be reached by using the function `getTranslationUnitDecl()`. This top level node contains the *identifier table*, which allows the user to navigate to the single nodes using pointers. Furthermore, the different nodes in *Clang* are only divided into three large classes, *declarations*, *statements*, and types, whereby each powerful core class contains a large subset of different sub-classes [15]. For instance, there is a large number of different types of *declarations* that are all grouped under the super-class *declarations*. The best known and most commonly used is the *declaration* of a variable. To give another example, *compound statements* and *binary operators* are grouped under the super-class *statements*. It should also be noted that there is no general superclass for all types of nodes, but that all nodes are assigned to one of the core classes. Since each core class has a different type of navigation, there is no possibility to visit all nodes equally. For instance, we can navigate through an `if`-*statement* by calling the functions `getThat()` or `getElse()`. In contrast, the function `getBody()` must be called to access the contents of a `while`- or a `for`-loop.

Figure 2.1 shows the *abstract syntax tree* of the simple *C++* program we introduced in Listing 1.1. The representation of the *abstract syntax tree* can be displayed by executing

the command `clang-check -ast-dump [Filename]` [16]. It can be seen that the topmost instance of the tree is represented by the node `TralsationUnitDecl`. In the first sub-level, we find the nodes of type `TypedefDecl`, which describe predefined types like integer or strings. Furthermore, *declarations* of functions are shown at this level. The node `CompoundStmt` groups all *statements* and expressions that are children of another node. If we look at the lower levels of the tree, we finally see the individual *statements* that are each represented by a node. If we take a closer look at the information provided for each node, the start location and the end location can be noted. The data about the location is stored in the so-called *source manager*, each pointing to a token in the programming code. To get these locations in a *Clang* application, the functions `getLocStart()` or `getLocEnd()`, which point to the first or the last token of the instruction, can be executed.

To enable the user to develop his own tool based on the *Clang* framework, the so-called `RecursiveASTVisitor` is provided. This contains a set of predefined functions and brings together all the information on how to access different *declarations*, *statements* and types. For instance, the method `VisiteStmt(void)` provides a routine to traverse all the *statements* in the code. Users can access these interfaces to implement their own logic and perform *source-to-source* transformations. *Clang*'s functions and existing infrastructure can thus be used to find places in an application and insert the desired code there [7]. Therefore, the framework is very well-suited for writing an application that satisfies the objectives of the thesis. For the development of the tool we will use *LLVM* version 14.0.1 and *Clang* version 14.0.1 built on top of it, also relying on the associated documentation [17].

```
TranslationUnitDecl 0x7f8872837808 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7f8872838100 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x7f8872837da0 '__int128'
|-TypedefDecl 0x7f8872838170 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7f8872837dc0 'unsigned __int128'
|-TypedefDecl 0x7f88728384e8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_tag'
| `-RecordType 0x7f8872838260 '__NSConstantString_tag'
|   `-CXXRecord 0x7f88728381c8 '__NSConstantString_tag'
|-TypedefDecl 0x7f8872838580 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7f8872838540 'char *'
|   `-BuiltinType 0x7f88728378a0 'char'
|-TypedefDecl 0x7f887480c4f8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag [1]'
| `-ConstantArrayType 0x7f887480c4a0 '__va_list_tag [1]' 1
|   `-RecordType 0x7f8872838670 '__va_list_tag'
|     `-CXXRecord 0x7f88728385d8 '__va_list_tag'
|-FunctionDecl 0x7f887480c598 </Users/maximilian/IntroExample.cpp:1:1, line:11:1> line:1:6 used loopFunc 'void ()'
| `-CompoundStmt 0x7f887480cb90 <col:17, line:11:1>
|   |-ForStmt 0x7f887480c7f8 <line:2:3, line:4:3>
|   | |-DeclStmt 0x7f887480c720 <line:2:9, col:18>
|   | | `-VarDecl 0x7f887480c698 <col:9, col:17> col:13 used i 'int' cinit
|   | |   `-IntegerLiteral 0x7f887480c700 <col:17> 'int' 0
|   | |-<<<NULL>>>
|   | |-BinaryOperator 0x7f887480c790 <col:20, col:24> 'bool' '<'
|   | | |-ImplicitCastExpr 0x7f887480c778 <col:20> 'int' <LValueToRValue>
|   | | | `-DeclRefExpr 0x7f887480c738 <col:20> 'int' lvalue Var 0x7f887480c698 'i' 'int'
|   | | `-IntegerLiteral 0x7f887480c758 <col:24> 'int' 10
|   | |-UnaryOperator 0x7f887480c7d0 <col:28, col:29> 'int' postfix '++'
|   | | `-DeclRefExpr 0x7f887480c7b0 <col:28> 'int' lvalue Var 0x7f887480c698 'i' 'int'
|   | `-CompoundStmt 0x7f887480c7e8 <col:33, line:4:3>
|   |-ForStmt 0x7f887480c9a8 <line:5:3, line:7:3>
|   | |-DeclStmt 0x7f887480c8d0 <line:5:9, col:18>
|   | | `-VarDecl 0x7f887480c848 <col:9, col:17> col:13 used i 'int' cinit
|   | |   `-IntegerLiteral 0x7f887480c8b0 <col:17> 'int' 0
|   | |-<<<NULL>>>
|   | |-BinaryOperator 0x7f887480c940 <col:20, col:24> 'bool' '<'
|   | | |-ImplicitCastExpr 0x7f887480c928 <col:20> 'int' <LValueToRValue>
|   | | | `-DeclRefExpr 0x7f887480c8e8 <col:20> 'int' lvalue Var 0x7f887480c848 'i' 'int'
|   | | `-IntegerLiteral 0x7f887480c908 <col:24> 'int' 10
|   | |-UnaryOperator 0x7f887480c980 <col:28, col:29> 'int' postfix '++'
|   | | `-DeclRefExpr 0x7f887480c960 <col:28> 'int' lvalue Var 0x7f887480c848 'i' 'int'
|   | `-CompoundStmt 0x7f887480c998 <col:33, line:7:3>
|   `-ForStmt 0x7f887480cb58 <line:8:3, line:10:3>
|     |-DeclStmt 0x7f887480ca80 <line:8:9, col:18>
|     | `-VarDecl 0x7f887480c9f8 <col:9, col:17> col:13 used i 'int' cinit
|     |   `-IntegerLiteral 0x7f887480ca60 <col:17> 'int' 0
|     |-<<<NULL>>>
|     |-BinaryOperator 0x7f887480caf0 <col:20, col:24> 'bool' '<'
|     | |-ImplicitCastExpr 0x7f887480cad8 <col:20> 'int' <LValueToRValue>
|     | | `-DeclRefExpr 0x7f887480ca98 <col:20> 'int' lvalue Var 0x7f887480c9f8 'i' 'int'
|     | `-IntegerLiteral 0x7f887480cab8 <col:24> 'int' 10
|     |-UnaryOperator 0x7f887480cb30 <col:28, col:29> 'int' postfix '++'
|     | `-DeclRefExpr 0x7f887480cb10 <col:28> 'int' lvalue Var 0x7f887480c9f8 'i' 'int'
|     `-CompoundStmt 0x7f887480cb48 <col:33, line:10:3>
`-FunctionDecl 0x7f887480cc80 <line:13:1, line:16:1> line:13:5 main 'int ()'
  `-CompoundStmt 0x7f887480ce40 <col:16, line:16:1>
    |-CallExpr 0x7f887480cdf0 <line:14:3, col:12> 'void'
    | `-ImplicitCastExpr 0x7f887480cdd8 <col:3> 'void (*)()' <FunctionToPointerDecay>
    |   `-DeclRefExpr 0x7f887480cd90 <col:3> 'void ()' lvalue Function 0x7f887480c598 'loopFunc' 'void ()'
    `-ReturnStmt 0x7f887480ce30 <line:15:3, col:10>
      `-IntegerLiteral 0x7f887480ce10 <col:10> 'int' 0
```

Figure 2.1: An Example of the *Abstract Syntax Tree* Provided by *Clang*.

CHAPTER 3

Identifying Regions of Interest through AST Analysis

The aim is to develop a model that can efficiently analyze the *abstract syntax tree* created by *Clang* for our type of application. The *ROIProfiler* is based on this concept. In order to be able to make modifications to a program code, we first have to take a closer look at the *abstract syntax tree* of the input application. The analysis will focus on identifying the different node classes provided by *Clang* and how they can be categorized for our use case. We also need to look at how the positions for the performance counters can be found in the representation and what further modifications are necessary to insert the counters at the correct location. Finally, we need to find out how the program flow can be structured and whether it is possible to develop a recursive model that can be applied to all classes.

3.1 Identification of the Core Classes

In *Clang*, all instructions are grouped into the superclasses *declarations*, *statements*, and types. The core classes, which contain definitions for *declarations* and *statements*, are particularly interesting for our field of application, since many basic functionalities of a program can be represented with the instructions described in these classes. We will not discuss subclasses of the core class types any further, since we do not require information about the data types of the instructions in our type of application. In the following the classes provided by the *Clang* framework must be grouped and it must be analyzed how they are handled by the *ROIProfiler*.

The *declarations* class contains, among others, the classes `FunctionDecl` and `VarDecl`. These classes are used to represent descriptions of functions and variables in the *abstract syntax tree*. Listing 3.1 shows a code snippet of a program in which a main function

Listing 3.1: Example Code Showing the Definition of Functions and Variables.

```
int main(void) {
    int integer = 1;
    char character = 'a';
    bool boolean = true;
    float floatingPoint = 1.1;
    double doubleFloatingPoint = 1.2;
}
```

```
Dumping main:
FunctionDecl 0x7f9e3704d580 </Users/FuncVarDeclASTExample.cpp:1:1, line:8:1> line:1:5 main 'int ()'
`-CompoundStmt 0x7f9e3704da18 <col:16, line:8:1>
  |-DeclStmt 0x7f9e3704d730 <line:2:9, col:24>
  | `-VarDecl 0x7f9e3704d6a8 <col:9, col:23> col:13 integer 'int' cinit
  |   `-IntegerLiteral 0x7f9e3704d710 <col:23> 'int' 1
  |-DeclStmt 0x7f9e3704d7d8 <line:3:9, col:29>
  | `-VarDecl 0x7f9e3704d758 <col:9, col:26> col:14 character 'char' cinit
  |   `-CharacterLiteral 0x7f9e3704d7c0 <col:26> 'char' 97
  |-DeclStmt 0x7f9e3704d878 <line:4:9, col:28>
  | `-VarDecl 0x7f9e3704d800 <col:9, col:24> col:14 boolean 'bool' cinit
  |   `-CXXBoolLiteralExpr 0x7f9e3704d868 <col:24> 'bool' true
  |-DeclStmt 0x7f9e3704d948 <line:5:9, col:34>
  | `-VarDecl 0x7f9e3704d8a8 <col:9, col:31> col:15 floatingPoint 'float' cinit
  |   `-ImplicitCastExpr 0x7f9e3704d930 <col:31> 'float' <FloatingCast>
  |     `-FloatingLiteral 0x7f9e3704d910 <col:31> 'double' 1.100000e+00
  `-DeclStmt 0x7f9e3704da00 <line:6:9, col:41>
    `-VarDecl 0x7f9e3704d978 <col:9, col:38> col:16 doubleFloatingPoint 'double' cinit
      `-FloatingLiteral 0x7f9e3704d9e0 <col:38> 'double' 1.200000e+00
```

Figure 3.1: The *Abstract Syntax Tree* for *Declarations* of Functions and Variables.

is defined that contains assignments to different primitive data types of *C++*. The corresponding *abstract syntax tree* representation of *Clang* is shown in Figure 3.1. The information contained in the `FunctionDecl` class is needed to get the content of a function. In the *ROIProfiler*, this data is used to get the content of the main function at the beginning of the analysis process and to handle function calls. If we look at the *declaration* of a variable in more detail, we notice that the variable *declaration* itself is grouped under the `DeclStmt` class. Since we do not need to distinguish between the different data types for the insertion of measurement code, it is sufficient to examine the higher-level class `DeclStmt` more closely. This class, however, is located in the core class *statements*.

The class *statements* also contains all the other classes that we need to specify more. In particular, we are interested in the classes `DeclStmt`, `OperatorCallExpr`, `BinaryOperator`, `CallExpr`, `CXXCallExpr`, `IfStmt`, `CaseStmt`, `ForStmt`, `DoStmt`, and `WhileStmt`. These classes were selected, since by describing these a variety of different programs can be analyzed by our transformation tool. These instructions provide basic functionalities which are used in each application and thus must be recognized by our profiler. We limit ourselves however in such a way that we do not have to identify each class which is made available by *Clang*, but address the most important classes mentioned before.

Listing 3.2: Example Code Showing the Definition of a Function Call.

```
void loopFunc() {
    /* code block */
}

int main(void) {
    loopFunc();      // CallExpr Class
}
```

```
Dumping main:
FunctionDecl 0x7faba602fac0 </Users/CallExprASTExample.cpp:5:1, line:7:1> line:5:5 main 'int ()'
`-CompoundStmt 0x7faba602fc50 <col:16, line:7:1>
  `-CallExpr 0x7faba602fc30 <line:6:2, col:11> 'void'
    `-ImplicitCastExpr 0x7faba602fc18 <col:2> 'void (*)()' <FunctionToPointerDecay>
      `-DeclRefExpr 0x7faba602fbd0 <col:2> 'void ()' lvalue Function 0x7faba602f8f8 'loopFunc' 'void ()'
```

Figure 3.2: The *Abstract Syntax Tree* for the `CallExpr` Class.

3.1.1 Leaf Nodes

A group of *statements* is to be formed that contains operations that cannot be traversed further. We will call these instructions *leaf nodes*. These include the frequently used classes `DeclStmt`, `OperatorCallExpr`, and `BinaryOperator`. Nodes designated with the `DeclStmt` class represent a variable *declaration*. This operation can be neglected in the analysis of programs, since the runtime can be equated with a constant value. `OperatorCallExpr` is used whenever an operator is called. For instance, this can occur when the software writes a value to a stream and outputs it. For nodes belonging to the `BinaryOperator` class, two operands are used to get a result through a calculation or comparison. All three classes are examples of actions that are executed only once and cannot be further broken down into smaller parts. The group also contains other classes provided by the *Clang* framework that have similar characteristics. This group has the property that the *ROIProfiler* does not have to further traverse the instructions, but that measurement code can be directly be wrapped around them.

3.1.2 Parent Nodes

We will define all other classes in which we are interested in as *parent nodes*. These classes have the characteristics of an underlying layer that can also be traversed. They are of particular importance because we need to handle each special case of them separately. The code for a simple method call can be seen in Listing 3.2. The corresponding representation in the *abstract syntax tree* can be seen in Figure 3.2, where the actual method call is specified by the class `CallExpr`. In addition to the information about the location, this node also contains a reference to the function that is called. When the tool found this type of node it is necessary to check whether other nodes exist on the same level. If this is the case, the *parent node* with all sub-levels can be enclosed as one unit with measurement code. If, however, there are no other siblings on this level, we must examine

Listing 3.3: Example Code Showing the Definition of a Branch *Statement*.

```
int main(void) {
    int sum = 2;
    if(sum > 1)          { /* code block */ }
    else if(sum > 2)     { /* code block */ }
    else                 { /* code block */ }
}
```

```
Dumping main:
FunctionDecl 0x7ff47b06b980 </Users/IfStmtASTExample.cpp:1:1, line:8:1> line:1:5 main 'int ()'
`-CompoundStmt 0x7ff47b06bb90 <col:16, line:8:1>
  `-IfStmt 0x7ff47b06bb60 <line:2:9, line:7:1> has_else
    |-ImplicitCastExpr 0x7ff47b06bab0 <line:2:12> 'bool' <IntegralToBoolean>
    | `-IntegerLiteral 0x7ff47b06ba90 <col:12> 'int' 1
    |-CompoundStmt 0x7ff47b06bac8 <col:14, line:3:1>
    `-IfStmt 0x7ff47b06bb30 <col:9, line:7:1> has_else
      |-ImplicitCastExpr 0x7ff47b06baf8 <line:3:12> 'bool' <IntegralToBoolean>
      | `-IntegerLiteral 0x7ff47b06bad8 <col:12> 'int' 2
      |-CompoundStmt 0x7ff47b06bb10 <col:14, line:4:1>
      `-CompoundStmt 0x7ff47b06bb20 <line:6:7, line:7:1>
```

Figure 3.3: The *Abstract Syntax Tree* for the `IfStmt` Class.

Listing 3.4: Example Code Showing the Definition of a Loop.

```
int main(void) {
    for(int i = 0; i < 10; i++) { /* code loop */ }
}
```

the subordinate level more closely. This is necessary because there would be no added value if only the runtime of a single possible resource-intensive class were output. In this case, we can use the `getDirectCallee()` function provided by *Clang* to get the function that is called. For the class `CXXMemberCallExpr` exactly the same approach can be applied.

The code of a nested `if-else`-query is shown in Listing 3.3, with the associated *abstract syntax tree* shown in Figure 3.3. The first `if`-*statement* represents the first query, whereas the nested `if`-*statement* is actually an `else-if`-branch. Both `if`-*statements* have the `has_else` flag, which means that there is an `else`-branch in the query construction. To get the body of the `if`-*statement* we use the `getThen()` command. Once we have reached the last `else-if`-branch, we check whether a flag labeled `has_else` is set and in this case we can execute a `getElse()` query. Nodes belonging to the Class `CaseStmt` can be traversed analogously to `if`-*statements* with the same method, but instead of the `getThen()` function we use the `getBody()` function.

The implementation of a `for`-loop, another *parent node* that needs to be analyzed, can be seen in Listing 3.4. The *abstract syntax tree* generated from the code can be seen in Figure 3.4. The analysis of loops is particularly complicated because the instructions in them can be executed several times. To get the content of the loop, we can simply use the function `getBody()`, but the runtime calculation of the performance counter

```
Dumping main:
FunctionDecl 0x7f8d6887eb80 </Users/ForStmtASTExample.cpp:1:1, line:5:1> line:1:5 main 'int ()'
`-CompoundStmt 0x7f8d6887ee40 <col:16, line:5:1>
  `-ForStmt 0x7f8d6887ee08 <line:2:2, line:4:2>
    |-DeclStmt 0x7f8d6887ed30 <line:2:7, col:16>
    | `-VarDecl 0x7f8d6887eca8 <col:7, col:15> col:11 used i 'int' cinit
    |   `-IntegerLiteral 0x7f8d6887ed10 <col:15> 'int' 0
    |-<<<NULL>>>
    |-BinaryOperator 0x7f8d6887eda0 <col:18, col:22> 'bool' '<'
    | |-ImplicitCastExpr 0x7f8d6887ed88 <col:18> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0x7f8d6887ed48 <col:18> 'int' lvalue Var 0x7f8d6887eca8 'i' 'int'
    | `-IntegerLiteral 0x7f8d6887ed68 <col:22> 'int' 10
    |-UnaryOperator 0x7f8d6887ede0 <col:26, col:27> 'int' postfix '++'
    | `-DeclRefExpr 0x7f8d6887edc0 <col:26> 'int' lvalue Var 0x7f8d6887eca8 'i' 'int'
    `-CompoundStmt 0x7f8d6887edf8 <col:31, line:4:2>
```

Figure 3.4: The *Abstract Syntax Tree* for the `ForStmt` Class.

inserted there has to be taken into account. This is problematic because the inserted measurement code is called as often as the loop itself. The times of individual instructions must therefore be added together so that the total time of a *statement* can be calculated. The problem of the overhead generation will be discussed in more detail in Chapter 5. The process of traversing `while`- and `do/while`-loops can be treated analogously to that of `for`-loops. It can thus be stated that *parent nodes* are only enclosed with performance counters if other elements exist on the same level. If this is not the case, the tool can immediately analyze the underlying level.

Furthermore, the *abstract syntax trees* in Figure 3.2, Figure 3.3, and Figure 3.4 show nodes with the label `CompoundStmt`. This is used to group children of a superordinate node. In the `CompoundStmt` provided by *Clang*, however, there are nodes from both of the groups we identified. Instructions that belong to the group of *leaf nodes*, including simple *declarations* of variables, should also be grouped together. To do this, however, classes that themselves contain further children must be removed and treated separately. For this purpose, we will create a separate class called `CustomCompoundStmt` when traversing. For each node traversed, the tool checks which group it belongs to. If the current node is a *leaf node*, it is added to a `CustomCompoundStmt` or a new `CustomCompoundStmt` is initialized. A node of type `CustomCompoundStmt` is closed when the first *parent node* is reached. After this class has been handled separately, a new `CustomCompoundStmt` can be initialized. This allows the runtime of all *leaf nodes* to be combined. It should also be mentioned that this grouping is not done at runtime of the input program, but already in the analysis process of the tool, which reduces the resource consumption of the tool.

3.1.3 Classification Approach by Example

The collected knowledge about the classes of the nodes can now be used to analyze the example code, shown in Listing 3.5, of an application that calculates the first hundred Fibonacci numbers. In order to better understand the procedure of the *ROIProfiler*, we will also start traversing the nodes in the main function. In the first line of the main function, a function call can be seen, which can be classified as a *parent node*, since the sub-tree of the function *declaration* can be accessed by calling this function. In the first

Listing 3.5: Fibonacci Application Example for Analysing the Identified Groups.

```
#include <iostream>

int fibonacci() {
    /* CustomCompoundStmt start */
    double n = 0;                                   // leaf node
    t1 = 0;                                         // leaf node
    t2 = 1;                                         // leaf node
    nextTerm = 0;                                   // leaf node
    n = 100;                                        // leaf node
    std::cout << "Fibonacci Series: ";              // leaf node
    /* CustomCompoundStmt end */

    for (int i = 1; i <= n; ++i) {                  // parent node
        if(i == 1) {                                // parent node
            /* CustomCompoundStmt start */
            std::cout << t1;                        // leaf node
            std::cout << ", ";                      // leaf node
            /* CustomCompoundStmt end */
        }
        else if(i == 2) {                           // parent node
            /* CustomCompoundStmt start */
            std::cout << t2;                        // leaf node
            std::cout << ", ";                      // leaf node
            /* CustomCompoundStmt end */
        }
        /* CustomCompoundStmt begin */
        nextTerm = t1 + t2;                         // leaf node
        t1 = t2;                                    // leaf node
        t2 = nextTerm;                              // leaf node
        std::cout << nextTerm << ", ";              // leaf node
        /* CustomCompoundStmt end */
    }
}

int main(void) {
    fibonacci();                                    // parent node
    return 0;
}
```

sub-level of the function *declaration*, there are first six instructions that can be marked as *leaf nodes*, since no further levels can be reached through these nodes. Furthermore, these instructions can be combined as `CustomCompoundStmt`, as the individual runtimes will be negligible. Next, there is a `for`-loop that must be declared as a *parent node*, since further instructions are called from it. The two `if`-instructions found in the level below also contain children and are therefore also defined as *parent nodes*. Each instruction contains two simple instructions that can declared as *leaf node* and grouped together as `CustomCompoundStmt`. At the end of the function there are four more instructions that are also marked as *leaf nodes* and can be combined. It should be noted that in the actual traversal process not all levels are annotated at once. If the *ROIProfiler* is executed without further options, only the content of the `fibonacci()` function would be annotated for the time being. With the information gained from the first run, the user can specify a desired region to be looked at in more detail.

Finally, it can be stated that the nodes of the *abstract syntax tree* can be divided into two groups for our purposes. *Leaf nodes* that are summarized as `CustomCompoundStmt` and *parent nodes* that have to be considered separately. For groups of *leaf nodes*, only one shared runtime is calculated by the *ROIProfiler*. *Parent nodes* are tracked individually if other instructions exist on the same level. If only one superclass is found on the currently examined level, its content is analyzed in more detail. If the special case occurs that each level contains only one instruction, only the total runtime of the program is calculated and output, as no detailed profiling statistic can be provided for the input application.

3.2 Detection of the Insert Locations

In order to insert the measurement code in the right places, we need information about the positions of the nodes in the actual program code. These are stored in the *source manager* of the *abstract syntax tree* and can be accessed by calling functions of the *Clang* infrastructure. For the *ROIProfiler*, functions that access the first and last location of the node are particularly important. However, the results of these predefined functions must be modified. If we now use the functions to obtain the boundary positions of a node and insert our measurement code into them, we get an output that looks similar to the code shown in Listing 3.6. It can be seen that the example code was inserted correctly before the instruction under consideration, but the end tag differs from the desired result. To generate code that creates the desired result, the code must be inserted after the closing curly brace. However, the locations obtained with the `getBeginLoc()` function can be used immediately to insert the code at these locations.

The function `getEndLoc()`, on the other hand, does not return the first place after the examined node, but the last place in it. As can be seen in Listing 3.6, this leads to the measurement code not correctly enclosing the node. If the instruction does not contain opening and closing brackets, the position before the semicolon may also be returned. Both problems can be solved analogously. The solution is to develop a function that checks the current token and moves the position with the information obtained. To do

Listing 3.6: Wrong Insertion Position When Using End Location of Node.

```
void loopFunc() {
    startEvent(1);          // getBeginLoc() returns the correct position
    while ( i <= 10 ) {
        i++;
        endEvent(1);        // getEndLoc() returns the incorrect position
    }
    /* "endEvent(1);" should be inserted at the position after the brace */
}
int main(void) {
    startEvent(0);
    loopFunc();
    endEvent(0);
    print();
    return 0;
}
```

this, first check whether the token at the current position is of type `clang::tok::semi` or of type `clang::tok::r_brace`, which allows the parameters for subsequent actions to be determined. Afterwards, the interface *Lexer* provided by *Clang*, which allows certain actions on the streams of tokens, can be used to find the desired position with the function `Lexer::FindLocationAfterToken()`. This method takes the current token and the character to be searched for and returns the position after the first character searched for. Finally, we can state that the start position and the end position of all nodes are now returned correctly for our purposes. With these positions, the measurement code can be wrapped around different groups of nodes.

In the following we discuss the flow of the *ROIProfiler* and explain how all the different types of nodes can be traversed recursively. For this purpose, the input parameters and the expected output of the *ROIProfiler* should be defined first. As a mandatory input parameter, the *ROIProfiler* should accept a source code written in *C/C++*. The storage location for the transformed program can be specified optionally. Furthermore, the user can decide in which *regions of interest* instructions are to be annotated. At the beginning of the profiling process, the *ROIProfiler* should wrap all code blocks of the main function with performance counters and return transformed software that can be executed by the user. In addition to the actual output of the input program, the user is provided with useful feedback on the resource consumption of individual instructions in the main function. Each instruction that supports a more detailed analysis is shown with an identification number. The user can utilize these identifiers to start the next iteration of the analysis by specifying desired regions and launching the *ROIProfiler* again.

Now that the basic characteristics of the *ROIProfiler* are known, the logic of the process behind the *ROIProfiler* can be analyzed. A flow diagram of the simplified process is shown in Figure 3.5. The process is started by calling the *ROIProfiler* with the correct parameters. The first step is to check which mode is currently active and, depending on

this, either search for the *entry point* to the input application or search for an instruction specified by the user. In both cases, the source code created by *Clang* is first searched for the desired node. The *ROIProfiler* is terminated if neither a main function nor a node with the searched identifier is found. If the search is successful, the node found of the instruction is passed to the `traverseChildren(Stmt*)` function. Further, the content of the passed node is taken so that the child nodes can be counted and categorized. The number of children is needed to make a basic case distinction. If several nodes are located at this level, they can be annotated and the program can be terminated. However, if only one node is found at this level, it must be checked whether it belongs to the group of *parent nodes*. The special case of only one instruction occurring in each level is achieved if the node is not a *parent node*. In this case, only the runtime of the entire application is returned, as no analysis of individual instructions is possible. However, if the *ROIProfiler* detects another level that can be analyzed, the content of the *parent node* can simply be forwarded recursively.

In the process shown in Figure 3.5, it can be stated that all properties for the development of a recursive method apply. In general, the overarching problem is to examine all levels of the *abstract syntax tree* and to assign performance counters to certain sub-areas. The sub-problem is to examine a single level of the tree and to examine nodes on this level. Furthermore, the *ROIProfiler* has two base cases that serve as termination conditions. One is met when multiple nodes at a level are found, in which case we can simply annotate them all and solve the problem. The other termination condition is given if only one *leaf node* exists which has no other levels below it. If there is only one instruction on a level that has other child nodes, we can simply traverse that level by calling the function itself. This procedure is repeated until either all levels have been traversed, or until a level has been found that can be annotated and the overall problem has been solved. This is very useful, since it is not necessary to consider each level separately, or even to count the depth of the *abstract syntax tree* at the beginning. Thus, it makes no difference how many levels are below the starting point, as the program only terminates after the last level has been reached.

3.3 Application of a Recursive Model to the Core Classes

3.3.1 Traversing Approach by Example

Based on this method, the example code shown in Listing 3.7 can now be analyzed. When the traversal process begins, the main function is searched for and checked to see how many instructions have been specified in it. In this case, the main function contains only one instruction of the type `CallExpr` (`loopFunc()`). Since a node was found that has subordinate levels, the contents of this can be passed on to the `traverseChildren(Stmt*)` function. Subsequently, the children of this node are also examined, but this time an instruction of the class `ForStmt` is present. This class also falls into the group of *parent nodes* and the content must therefore be traversed again. On the next level, it is noticeable that not only one *leaf node* is to be executed. The termination condition is reached

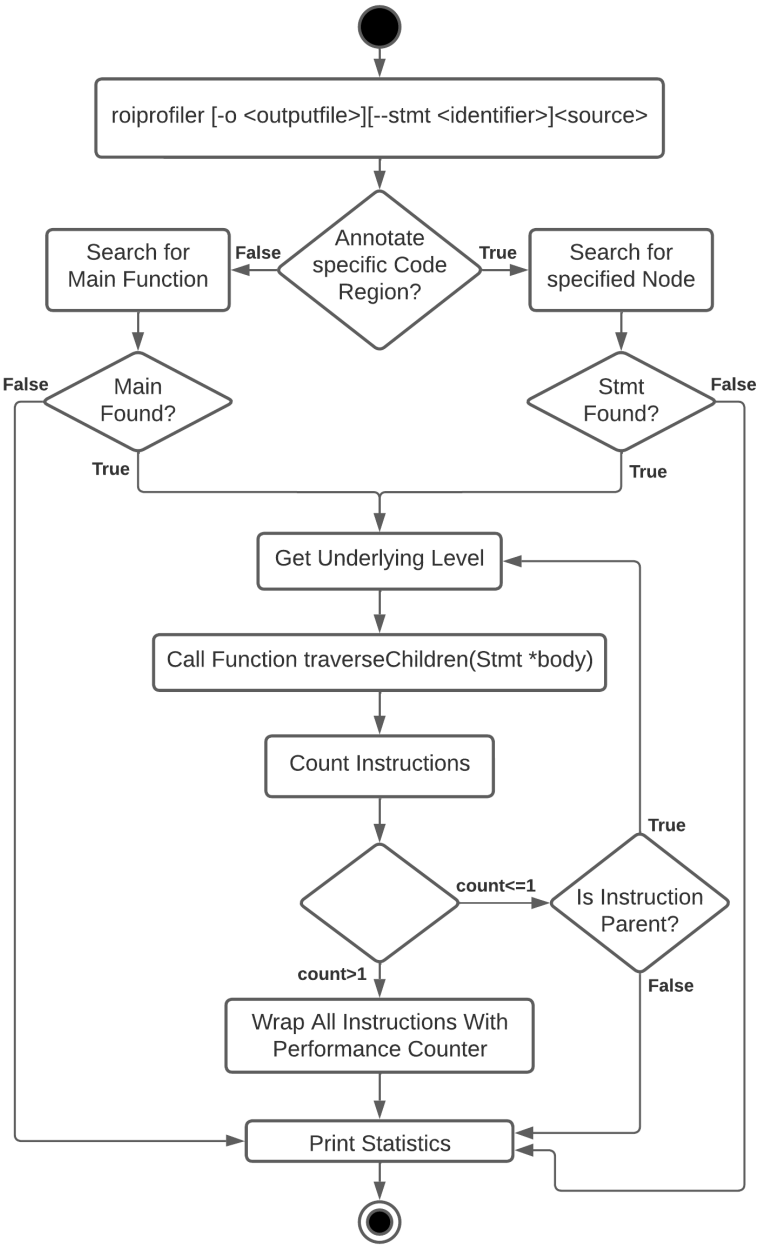

Figure 3.5: Flow Chart Representing the Logic of the *ROIProfiler*.

Listing 3.7: Example of the Model Used to Traverse the *Abstract Syntax Tree*.

```
void loopFunc() {
    // children.size() <= 1 && child.getClass() == Stmt::ForStmtClass
    // // => traverseChildren(child.getBody())
    for (int i = 0; i < 10; i++) {
        // children.size() > 1
        // => annotate all instructions
        for { /* code loop 1 */ }
        for { /* code loop 2 */ }
        for { /* code loop 3 */ }
    }
}

int main(void) {
    // children.size() <= 1 && child.getClass() == Stmt::CallExpr
    // => traverseChildren(child.getBody())
    loopFunc();
}
```

and all nodes on this level can be wrapped with performance counters. If we change the example a little and give the tool the identifier of the `for`-loop in the `loopFunc()` function as the starting point, all previous steps would be skipped by the *ROIProfiler* and the body of the `for`-loop can be annotated immediately. With the insertion of the performance counters, the first run of the profiling process is finished, and the user now has the option of running the *ROIProfiler* again with the identifiers of the `while`-loops to start a more detailed analysis of them.

Finally, it can be stated that a model was found with which all classes of the group of *parent nodes* can be treated equally. After the categorization of the nodes, depending on the class type, it can be determined how the underlying `CompoundStmt` class can be accessed. If access is gained to this node, it can be passed on as a node of the superclass `Stmt` as a parameter to the `traverseChildren(Stmt*)` function. The fact that the node can be represented as a `Stmt` is particularly important, as all the functions necessary for our purpose can be accessed. From this point on, the procedure is the same for all classes. It can therefore be said that when applying this model, it makes no difference which type of *parent node* is to be traversed. This is very useful, as a single recursively defined function is sufficient to traverse the sub-levels of all class types.

CHAPTER 4

Source-to-Source Transformation and Outcome Presentation

Clang provides another library called *LibTooling* which, in addition to accessing the *abstract syntax tree*, also provides predefined functions and classes that can be used to traverse a program structure and perform a *source-to-source* transformation. In Chapter 3 we looked at analyzing the structure of a program in which performance counters are to be inserted. In addition, the fundamental procedure of the profiling process was discussed. Thus, we have analyzed all the theoretical considerations and laid the foundation for the development of the *ROIProfiler*, which is the focus of this Chapter. For this purpose, we first look at which actions are provided by *Clang* and how these can be used to write our own application. Furthermore, we explore how the measurement code can be inserted efficiently and how the overhead can be reduced by performing resource-intensive operations before the actual execution of the application to be profiled. For the calculation of the runtimes and the output of the data, a library is introduced that can be customized by the user. Finally, we discuss the workflow of the profiling process.

4.1 Utilization of the Clang Infrastructure

In addition to the *abstract syntax tree* provided, which is enormously valuable for our purpose, *Clang* also provides a library that allows users to develop their own front-end compiler actions [18]. Since some of the functions provided by the library simplify the development of our *source-to-source* transformation application, we will use them. In the previous section, we already discussed the basic characteristics of the *ROIProfiler*. These can now be specified in more detail, so that the basic structure of the application can be created. In Listing 4.1, it can be seen that we give the tool the name *ROIProfiler* by creating an `llvm::cl::OptionCategory`.

Listing 4.1: Code Showing Option Definitions for Creating a *Clang* Tool.

```
/* Definition of the Tool Name */
static llvm::cl::OptionCategory MyToolCategory( "ROIProfiler options" );

/* Specification of the Output File Option */
static cl::opt <string> OutputFile( "o",
           cl::desc( "Write transformed file to custom location" ),
           cl::value_desc( "output file" ),
           cl::cat( MyToolCategory ));

/* Specification of the Statement Option */
static cl::opt <string> Statement( "stmt",
    cl::desc( "Specifies the current traversal point" ),
    cl::value_desc( "id of stmt" ),
    cl::cat( MyToolCategory ));


int main( int argc, const char **argv ) {
    /* Bind the Specified Options to the Tool */
    auto ExpectedParser = CommonOptionsParser::create(
                              argc, argv,
                              MyToolCategory,
                              llvm::cl::Required );
}
```

Furthermore, we will create two variables of the class `cl::opt`, which represent the parameters of the *ROIProfiler*. It is necessary to mention that we do not have to explicitly specify that one of the input parameters is a source code of an application, as all *LibTooling* tools require this property. However, we do need to indicate that the output location may be set with the `-o` option. Furthermore, we specify that the starting point of the traversal can be set by the user with the option `--stmt`. The final step is to create a common option parser in the main method of the tool, which contains the specified options. Once an executable program has been created from the code, it is possible to run it in the *Clang* environment, as shown in Figure 4.2.

After the basic properties of the program have been defined and the *ROIProfiler* has been registered in the *Clang* environment, it is now possible to consider how a front-end action can be built with the infrastructure provided. The *entry point* for this is the `FrontendAction` interface, with which custom actions can be executed on the code. A new `FrontendAction` can be implemented using the `Tool.run(newFrontendActionFactory)` instruction. Then the classes provided by *Clang* can be overwritten. The *C++* syntax for including the framework provided by *Clang* is shown in Listing 4.3. Next, the created front-end action is used to create an `ASTConsumer` for each traversion unit. This allows the *abstract syntax tree* to be accessed programmatically. For our scope, we will now call the function `TraverseDecl` in the class `Visitor`. We give this function the top level node of the *abstract syntax tree* as a parameter, which allows us to de-

Listing 4.2: The Synopsis of the *ROIProfiler*.

```
sh-3.2# ROIProfiler -help
USAGE: ROIProfiler [options] <source0> [... <sourceN>]

OPTIONS:

Generic Options:

--help                          - Display available options
--help-list                     - Display list of available options
--version                       - Display the version of this program

ROIProfiler options:

--extra-arg=<string>            - Additional argument to append to the compiler command line
--extra-arg-before=<string>     - Additional argument to prepend to the compiler command line
-o=<output file>                - Write transformed file to custom location
-p=<string>                     - Build path
--stmt=<id of stmt>             - Specifies the current traversal point
```

termine the *entry point* of the traversal. By calling this function, all nodes of the *abstract syntax tree* are visited. Finally, the `RecursiveASTVisitor` can be used to visit selected nodes of different classes. In the constructor of the `Visitor`, a reference to the *source manager* must be created, this will be important later for inserting the measurement code. The class `RecursiveASTVisitor` can utilize a variety of different functions to traverse the different node types. For our purpose, we need the functions `VisitFunctionDecl(FunctionDecl *func)` and `VisitStmt( Stmt *stmt )`. The former iterates over all definitions of functions in the code, which is needed to annotate the instructions in the main function if no `--stmt` option is given, and to insert performance counters that measure the runtime of the entire application. The `VisitStmt( Stmt *stmt )` function is needed to look for *statements* in the code that have been specified by the user. From this point on, the recursive model developed can be used to traverse each level of the *abstract syntax tree*.

4.2 Insertion of Callback Functions

The in Chapter 3 gathered knowledge about locating the different regions can now be used to insert measurement code. At the first accessible point in the source code, we will import the library needed to calculate the runtimes. Furthermore, the self-developed library *DataStorage*, which will be described in more detail later, is imported and initialized. All nodes for which performance indices are to be measured must be enclosed with performance counters. For each region, two positions are needed, one before and one immediately after the region. Before each region, the function `startEvent(int identifier)` should be called, and after it, the function `endEvent(int identifier)` should be called. To transform the input application, the interface `Rewriter` can be used. If we initialize this class, we get the possibility to use the function `InsertText(SourceLocation Loc, StringRef Str)` to modify the source code. Since we already know the correct locations of the instructions, we only need to make sure that they are forwarded to the function. In addition to the

Listing 4.3: Code Framework for Accessing the *Abstract Syntax Tree* Provided by *Clang*.

```
/* Definition of the RecursiveASTVisitor provided by Clang */
class Visitor : public RecursiveASTVisitor<Visitor> {
private:
    ASTContext *astContext;
public:
    explicit Visitor( ASTContext *Context ) : astContext( Context
    {
        rewriter.setSourceMgr( astContext->getSourceManager( ),
                               astContext->getLangOpts( ));
    }

    /* Functions for traversing the AST */
    virtual bool VisitStmt( Stmt *stmt ) {}
    virtual bool VisitFunctionDecl( FunctionDecl *func ) {}
    virtual ~Visitor( ) { }
};

/* Definition of the AST Consumer */
class Consumer : public ASTConsumer {
private:
    Visitor Visitor;

public:
    explicit Consumer( ASTContext *Context ) : Visitor( Context ) { }

    virtual void HandleTranslationUnit( ASTContext &Context )
    override { Visitor.TraverseDecl( Context.getTranslationUnitDecl( )); }
};

/* Definition of our frontend action */
class ClangFrontendAction : public clang::ASTFrontendAction {
public:
    virtual std::unique_ptr <clang::ASTConsumer>
    CreateASTConsumer( clang::CompilerInstance &Compiler,
                       llvm::StringRef InFile )
    override { return std::make_unique<Consumer>(
               Compiler.getASTContext( )); }
};

int main( int argc, const char **argv ) {
    /* Creation of a new frontend action */
    int result = Tool.run(
                    newFrontendActionFactory<ClangFrontendAction>( ).get( ));
}
```

position in the source code, the code to be inserted must also be specified. This function is always called when the position before or after a node to be annotated is reached. If a single node, such as a *parent node* or a stand-alone *leaf node*, is to be annotated, the start and end positions can simply be determined. The `startEvent(int identifier)` function is then inserted at the start position, and the `endEvent(int identifier)` function at the end position. If several nodes are combined into a `CustomCompoundStmt`, the start position is noted when the first node is visited and the end position when the last node is reached. The two functions can now be inserted at the positions found. Furthermore, it should be noted that the entire main class is enclosed with measurement codes and, if necessary, the runtime of a selected scope is also measured. Finally, after the last *statement* in the main class, a function `print()` is added, which is responsible for the output of the collected measured values.

Listing 4.4 shows an example of how the transformed code looks with the measurement code inserted. It can be seen that the *DataStorage* library is included and can already be filled with data from the annotated nodes during initialization. In this case, both the total runtime and the runtime of the desired region are measured. Within the region there are four *leaf nodes*, whereby two can be combined in a `CustomCompoundStmt` class. Furthermore, the three `while`-loops are annotated. Finally, the `print()` method is called as the last *statement*, which outputs the collected data in a table. Thus, it can be stated that all functions needed for the calculation of the performance have been inserted in the correct places.

The *DataStorage* library was primarily developed to keep the generated source code clean, but it also offers the user the possibility to build in their own logic for performance analysis. By placing the actual instructions in a separate file, only a small number of short instructions need to be added to the source code of the application, thus retaining readability. The library is included in every application transformed with the *ROIProfiler* and the *DataStorage* can be filled with resource-intensive information about the nodes at the beginning. This procedure is made possible because the inclusion and initialization of this library can be performed as the final step in the transformation process. The biggest advantage is that an array with known memory allocation can be created at the beginning of the program code, into which all resource-intensive information about the annotated nodes can be statically inserted. For each annotated node, a character string for the identification and the type of the node is inserted in the array. This can massively reduce resource consumption, as the `startEvent(int identifier)` and `endEvent(int identifier)` functions only require a numerical value with which the current event can be identified. The reference is then made using the index of the array, which means that only one access to the memory is required for each measured value. It should also be noted that the library is not initialized in the main class, but after the `include`-*statements* section, which means that the calculations in this section have no effect on the profiling process, as this section is never wrapped with performance counters. In the following, we will go into more detail about the *DataStorage* class and the logic we used for the runtime measurement.

Listing 4.4: Example Code for Inserting Callback Functions.

```
/* Include self-written library for data storage */
#include "../lib/DataStorage.cpp"

/* Initialization of the runtime array */
DataStorage dataStorage("Runtime", "Scope",
                        "CustomCompoundStatement i000001",
                        "WhileStmt 2085536",
                        "WhileStmt 2085537",
                        "WhileStmt 2085538",
                        "CustomCompoundStatement i000002" );

void foo () {
    dataStorage.startEvent(2)                    // start time of i000001
    int i = 1;
    i = i * 4;
    dataStorage.endEvent(2)                      // end time of i000001

    dataStorage.startEvent(3)                    // start time for 2085536
    while ( i <= 10 ) { /* code loop 1 */ }
    dataStorage.endEvent(3)                      // end time for 2085536

    dataStorage.startEvent(4)                    // start time for 2085537
    while ( i <= 20 ) { /* code loop 2 */ }
    dataStorage.endEvent(4)                      // end time for 2085537

    dataStorage.startEvent(5)                    // start time for 2085538
    while ( i <= 30 ) { /* code loop 3 */ }
    dataStorage.endEvent(5)                      // end time for 2085538

    dataStorage.startEvent(6)                    // start time for i000002
    i = i - 3;
    std::cout << i;
    dataStorage.endEvent(6)                      // end time for i000002
}

int main(void) {
    dataStorage.startEvent(0)                    // start time total runtime
    dataStorage.endEvent(1)                      // start time scope runtime
    foo();
    dataStorage.endEvent(1)                      // end time scope runtime

    /* instructions outside the scope */

    dataStorage.endEvent(0)                      // end time total runtime
    dataStorage.print();                         // print statistics
    return 0;
}
```

4.3 Measurement of the Performance Counters

We have already discussed the *DataStorage* library, which is a powerful extension to the *ROIProfiler* as it allows the user to insert their own measurement code. For our application, we need a second library called *chrono*, which is natively supported by *C++* [19]. The library provides functions for time measurement. In contrast to other libraries, the user is given the possibility to choose the level of accuracy, whereby for the application area of performance analysis, the times should be calculated as precise as possible.

The *DataStorage* class provides four basic functions. The first is the constructor, where all identifiers are stored in a data structure. For this, the string that is received as input is delimited at the separators and for each annotated node an entry is stored in the `StatementRuntime` array. Each entry contains an identifier, the type of the node as well as a start and a finish time. If the array is initialized, all known information can already be inserted. Furthermore, the start and end times are set to an invalid time. To prevent various errors in the time calculation, such as the *Y2262 problem*, in which the 64-bit integer used to count the nanoseconds is overflowed at a certain point in time, we use the first valid timestamp for this. The timestamps set during the runtime of the input application can be assigned to a node using the array index. This is possible because the complete code has already been analyzed during the *source-to-source* transformation.

The next function to be discussed is the `startEvent(int identifier)` function. This should be called before each instruction and should measure the current time. The `endEvent(int identifier)` function analogously determines the current time after the *statement*. However, for loops it must be noted that the sum of the runtimes of the functions in a loop should be counted. For this purpose, when the start event function is called, it is checked whether a time has already been entered for this *statement*. If this is the case, the runtime of the previous call is subtracted from the current time. In addition to measuring the time values, the functions also store how often they are called. The user thus receives important data about the accesses to a function, the number of loop passes and how many callback functions were inserted by the *ROIProfiler* in total.

Finally, a `print()` function is provided that can output all collected information to the command line. Note that this function is only executed after the last measurement and calculations performed at this time do not generate overhead. In this function, the total runtime of the application is calculated first. If a code *region of interest* selected by the user was computed, the total runtime of this region is also measured. A measurement evaluation is created for each estimated timestamp pair of a region. In addition to the calculated runtime, this structure also contains a character string that provides information about the region and is also to be output. It also stores how much time this region consumes compared to the total or the running time of a selected area. In addition to calculating these values, the `print()` function performs other actions such as calculating a suitable time format and converting the obtained runtimes into it.

4.3.1 Runtime Calculation by Example

Listing 4.5 shows an example of runtime calculations for a simple application that counts from zero to nine. In this example, the methods `startEvent(int identifier)` and `endEvent(int identifier)` have been replaced by the actual calculation to clarify which necessary operations are needed to measure the runtimes. The current time is read before and after an instruction with the function `high_resolution_clock::now()`. In this example, six measured values are required, as a total of two instructions and the entire running time of the application are to be determined. After all timestamps have been successfully read, the total duration must be calculated first. This is achieved by subtracting the start time from the end time, which results in a variable of type `chrono::duration<double, std:nano>`. This duration is stored in a double precision floating point in nanoseconds in order to calculate the time periods as precisely as possible. The computed duration can now be used to determine which unit suits best. After all calculations have been performed to the highest degree of accuracy, all runtimes are converted into this suitable time unit for output. After the individual runtimes have been calculated, they are compared to the total time span and the duration of the scope. Finally, it is possible to calculate how many resources are consumed by reading and writing the timestamps. For this purpose, the sum of the runtimes of the individual instructions can simply be subtracted from the duration of the scope. The output of the data collected is discussed in the following section.

4.4 Presentation of the Collected Data

When outputting the information obtained, a distinction can be made between the report of the *ROIProfiler* and the report of the profiling statistics. The output of the *ROIProfiler* itself can be seen in Listing 4.6. The first two values show the input file and the output file, whereby it should be noted that the output file in particular contains important information about the storage location if the user does not specify a user-defined location. The mode indicates whether an instruction to be traversed has been specified or if the run has been started in the main method. Next, it indicates how long the traversal and transformation took. Finally, it is indicated whether the *source-to-source* transformation was successful. This is not the case if an user-defined *statement* could not be found or if the code contains fatal errors that make compiling impossible. If the output is that the transformation was successful, the output file can now be compiled and executed so that the actual program can be profiled.

A table with information about the runtime is added to the actual output of the program. The output can be seen in Table 4.1. The first column contains the identifier of the code region, which can be used to re-run the *ROIProfiler* with the `--stmt` option to profile this segment in detail. The second column contains the class type of the node, which can be used to find the section in the source code more easily. The next column shows the runtime of an area, which is converted into a suitable time unit for different durations. For this purpose, the entire runtime is considered and, depending on the

Listing 4.5: Example Code for Measuring and Calculating Time Values.

```
DataStorage runtimes("Runtime",
                     "CustomCompoundStatement i000001",
                     "WhileStmt 1052237" );

void foo() {

    runtimes[1].startTime = high_resolution_clock::now()    //   5 ns
    int i = 0;
    runtimes[1].endTime   = high_resolution_clock::now()    //  15 ns

    runtimes[2].startTime = high_resolution_clock::now()    //  20 ns
    while (i < 10) {
        i++;
    }
    runtimes[2].endTime   = high_resolution_clock::now()    //  95 ns

}

int main(void) {

    runtimes[0].startTime = high_resolution_clock::now()    //   0 ns
    foo();
    runtimes[0].endTime   = high_resolution_clock::now()    // 100 ns

    /* calculation of the total runtime (100 ns - 0 ns = 100 ns) */
    total = runtimes[0].endTime - runtimes[0].startTime;

    /* calculation of the runtimes (end time - start time = runtime) */
    for each runtime (id stating at 1) {
        runtimes[id] = runtimes[id].endTime
                     - runtimes[id].startTime;
    }

    print(runtimes);
    return 0;
}
```

Table 4.1: Statistics Generated by the *ROIProfiler*.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
2083316	ForStmt	77.000 µs	0.35 %	0.35 %	1
2083741	ForStmt	109.000 µs	0.50 %	0.50 %	1
2084167	ForStmt	90.000 µs	0.41 %	0.41 %	1
2084593	ForStmt	94.000 µs	0.43 %	0.43 %	1
2085019	ForStmt	21 080.000 µs	97.04 %	95.98 %	1
Overhead		274.000 µs	1.26 %	1.25 %	12
Runtime		29 647.000 µs			

Listing 4.6: The Command Line Output of the *ROIProfiler*.

```
/* Execution of the ROIProfiler */
roiprofiler PerformanceTest.cpp -o PerformanceTestTransformed.cpp --stmt 1234

/* Statistics generated for the transformation phase */
Input File: PerformanceTest.cpp
Output File: PerformanceTestTransformed.cpp
Mode: Annotating Stmt 1234
Runtime: 955.13ms
Success: Yes

/* Compilation of the transformed file */
clang++ PerformanceTestTransformed.cpp -o CompiledProgram

/* Execution of the transformed program */
./CompiledProgram

/* Actual program output */

/* Profiling statistics (Table 4.1) */
```

length, a suitable unit is selected into which all times are converted. Possible time units are microseconds, milliseconds, and seconds. The fourth column shows the ratio of the scope running time to the time of the superordinate level, which can be calculated in two ways. If the user specifies the starting point of the traversal, the scope runtime is measured by inserting measurement code around this node. If no starting point is specified, the sum of all running times of the instructions is used. In comparison, the total runtime is always measured as the time between the first entry into the main function and the last point before termination of the application. The ratio of the resources spent on the total resources used can be seen in the next to last column of the table. The last column shows how often the performance counter pairs surrounding the *statement* were executed. In Table 4.1, it can be seen that the `for`-loops were executed only once. Note that these call values refer to the outer nodes and not to the iterations within a loop. After all instructions have been written, the amount of overhead generated by the performance counter is also calculated. The information about this can always be found in the second to last line of the table. In Table 4.1, it can be seen that the call value is listed as twelve. This value refers to the added performance counter. To calculate the runtime of the `for`-loops two counters each were needed, resulting in a total number of ten counters added. Furthermore, two performance counters were needed to measure the total runtime, resulting in a total of twelve counters. The last line of the table always displays the total required runtime.

With the data provided, the user can now see which instructions or areas of the code have the most load. If a particularly resource-consuming area is found, the user can locate it in the code. Furthermore, the identifier can be used to call the *ROIProfiler*

again with the `--stmt` option. In this case, the entire process can be repeated, whereby the user now receives more detailed information about a *region of interest*. Finally, it should be noted that this output is produced by the `print()` function in the *DataStorage* library. Thus, this function can also be modified for own purposes.

CHAPTER 5

Evaluation

In Chapter 3, we have shown which basic concepts have been used, and in Chapter 4, it was shown how this concepts can be applied to develop the *ROIProfiler*. In order to demonstrate that the goals regarding the included functionality have been achieved and also to show that the use of the tool is time efficient, a detailed analysis will be carried out. The first step is to check if the *ROIProfiler* provides the desired functionality and whether it can be applied to different applications. For this purpose, four programs of different complexity levels will be analyzed and transformed with the *ROIProfiler*. The programs created can then be analyzed. The performance of the *ROIProfiler* is to be tested by frequently running four applications in different environments. This allows the time of a program without performance counters to be compared with a program to which they have been added.

5.1 Functional Analysis

This section aims to demonstrate the effectiveness of the approach by showing the measurement results available through the application of the *ROIProfiler*. To show this, three programs of varying complexity are selected. Each application is analyzed with the *ROIProfiler* until a code region with a high resource consumption is found. The results of the measurements are presented and discussed for each hierarchy. In Section 5.2, the overhead generated by inserting measurement code into the benchmark programs is discussed in detail.

5.1.1 Varying Loop Runtimes

As a first example, let us analyze the code framework shown in the introduction. For this purpose, the basic structure is extended with several instructions, so that the code looks similar to the code shown in Listing 5.1. It can be seen that three `for`-loops are executed

Listing 5.1: *C++* Code Showing the *Varying Loop Runtimes* Application.

```
#include <iostream>
#include <vector>

int main(void) {
    for ( int i = 1; i <= 1000; i++ ) {
        std::cout << "Hello World";
    }
    for ( int i = 1; i <= 1000; i++ ) {
        int numberA = 892346;
        int numberB = 384378;
        int numberC = numberA * numberB;
    }
    for ( int i = 1; i <= 1000; i++ ) {
        std::vector<std::string> shoppingList;
        shoppingList.push_back("Milk");
        shoppingList.push_back("Eggs");
        shoppingList.push_back("Meat");
        shoppingList.push_back("Water");
        shoppingList.push_back("Sugar");
        shoppingList.push_back("Flour");
        shoppingList.push_back("Salt");
        std::sort( shoppingList.begin( ), shoppingList.end( ));
    }
    return 0;
}
```

in the main function, each loop performing a simple task a thousand times. The first loop writes a string to the command line, the second performs a multiplication of two numbers, and the last adds items to a shopping list and then sorts them.

The first step is to use the *ROIProfiler* to add measurement code to the source application. This requires executing the instructions shown in Listing 5.2. Initially, the *ROIProfiler* is called with the source code as input. The *ROIProfiler* analyzes the code, adds performance counters around the *regions of interest*, and returns a modified *C++* application. The program created can then be compiled into an executable application. The last step is to execute the created software. Since these steps work the same for all programs, we will not specify them separately.

The transformed program writes the results of the tasks and the statistics of the running times, which can be seen in Table 5.1, to the command line. The data provided shows at a glance that the last loop, which creates and sorts shopping lists, consumes the most amount of time. Furthermore, it can be seen that the first loop requires the second most resources and that the runtime of the second loop is almost negligible. If the user wants to improve performance based on this data, priority should be given to the level below the third loop.

Listing 5.2: Instructions for Transformation, Compilation and Execution.

```
/* source-to-source transformation with the roiprofiler */
roiprofiler VaryingLoopRunTimes.cpp

/* compilation of the generated code */
clang++ VaryingLoopRunTimes.cpp -o application

/* execution of the binary file */
./application
```

Table 5.1: Runtime Evaluation for the *Varying Loop Runtimes* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
2192956	ForStmt	402.291 µs	4.62 %	4.62 %	1
2193096	ForStmt	2.671 µs	0.03 %	0.03 %	1
2225085	ForStmt	8303.364 µs	95.34 %	95.34 %	1
Overhead		0.626 µs	< 0.01 %	< 0.01 %	12
Runtime		8709.478 µs			

Table 5.2: Runtime Evaluation for the *Fibonacci Sequence* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
i000002	CustomCompoundStmt	23.649 µs	2.29 %	2.29 %	1
2086033	ForStmt	1007.385 µs	97.70 %	97.66 %	1
Overhead		0.108 µs	0.01 %	0.01 %	6
Runtime		1031.509 µs			

5.1.2 Fibonacci Sequence

As a next example we want to analyze the *Fibonacci Sequence* program that calculates the first thousand numbers of the Fibonacci sequence. The code of the application is shown in Listing 5.3. We will follow the same process as in transforming the previous program to obtain the statistics shown in Table 5.2.

It can be seen that the `for`-loop consumes almost all of the runtime, so in this case it is not clear which region is to be revised. Therefore, the *ROIProfiler* must be called again with the parameter `--stmt 2086033`, which results in the statistics shown in Table 5.3. In this output, the higher runtime is noticeable, which arises because a calculation of the runtime must be carried out for each loop cycle. In total, the execution of the program now took 500 µs longer, whereby it can be seen that about half of the overhead occurred could be recognized by the *ROIProfiler* and shifted to a separate area. This occurs because a *declaration* is made first and only then the current time is written

Listing 5.3: *C++* Code Showing the *Fibonacci Sequence* Application.

```
#include <iostream>

int main(void) {
    double n, t1 = 0, t2 = 1, nextTerm = 0;
    n = 1000;

    std::cout << "Fibonacci Series: ";

    for (int i = 1; i <= n; ++i) {
        if(i == 1) {
            std::cout << t1;
            std::cout << ", ";
        }
        else if(i == 2) {
            std::cout << t2;
            std::cout << ", ";
        }
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;

        std::cout << nextTerm << ", ";
    }

    return 0;
}
```

to memory. Thus, the overhead caused by measuring the start time of an instruction can be eliminated, but not the time required to declare the end time. This occurrence will be discussed in more detail in the overhead analysis in Section 5.2. Looking at the table again, it can be seen that the `CustomCompoundStmt` takes up about seventy-five percent of the runtime and the instructions inside it are executed a thousand times. The reason for the high runtime consumption is that the calculations in both the `if`- and the `else-if`-branch are only executed once, while all other instructions are executed every time. Furthermore, it is noticeable that a scope runtime is now specified that contains the duration of all instructions below the user-specified *region of interest*. In this application, it was also possible to find a region that has a high resource consumption.

5.1.3 Password Generator

Next, a slightly more resource-intensive application will be tested, whereby the aim will also be to find the instruction that consumes the longest amount of time. For this purpose, the *Password Generator* program shown in Listing 5.4 will be analyzed in more detail. The program generates one million passwords, each containing three sections of six characters. After all passwords have been generated, the list is sorted and duplicates are

Table 5.3: Runtime Evaluation for the `ForStmt` of the *Fibonacci Sequence* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
2085332	IfStmt	140.928 µs	9.27 %	9.13 %	1000
i000004	CustomCompoundStmt	1153.832 µs	75.91 %	74.71 %	1000
Overhead		225.283 µs	14.82 %	14.59 %	4004
Scope		1520.043 µs			
Runtime		1544.359 µs			

Table 5.4: Runtime Evaluation for the *Password Generator* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
i000002	CustomCompoundStmt	< 0.001 ms	9.27 %	< 0.01 %	1
2219672	ForStmt	988.027 ms	75.91 %	16.07 %	1
i000005	CustomCompoundStmt	5161.804 ms	9.27 %	83.93 %	1
Overhead		< 0.001 ms	14.82 %	< 0.01 %	8
Runtime		6149.833 ms			

Table 5.5: Runtime Evaluation for the `ForStmt` of the *Password Generator* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
i000002	CCompoundStmt	1857.092 ms	20.11 %	12.55 %	1,8E+07
2219672	IfStmt	1556.085 ms	16.85 %	10.51 %	1,8E+07
i000005	CCompoundStmt	1529.157 ms	16.56 %	10.33 %	1,8E+07
Overhead		4292.683 ms	46.48 %	29.00 %	1,08E+08
Scope		9235.017 ms			
Runtime		14 800.639 ms			

eliminated. The sorted passwords are finally displayed to the user on the command line. In this example, it is particularly interesting to see if the generation of the passwords or the actions on the data structure require more time.

First of all, an overview of the running times of the first level should be generated, which can be seen in Table 5.4. In these statistics, the different runtimes of the various parts of the application can be seen very clearly. It is noticeable that the operations performed on the list require about five times the amount of time it takes to create one thousand passwords.

Table 5.5 shows the statistics of the analysis of the innermost for loop. It can be seen that the added performance counters consume more runtime than the actual instructions. This is due to the fact that six times per loop pass are measured, resulting in a total of 108 million *measurement values* recorded.

Listing 5.4: *C++* Code Showing the *Password Generator* Application.

```
#include <string>
#include <vector>
#include <iostream>

int main(void) {
    std::vector <std::string> passwordStorage;
    std::string password;
    char current;

    for (int passwordCount = 0; passwordCount < 1000000; passwordCount++) {
        for (int sectionCount = 0; sectionCount < 3; sectionCount++) {
            for (int sectionLength = 0; sectionLength < 6; sectionLength++) {
                char lowerCase = rand() % 26 + 'a';
                char upperCase = rand() % 26 + 'A';
                char number = rand() % 10 + '0';

                char randomChoice = (rand() % 100 + 1);
                if (randomChoice <= 40) {
                    current = lowerCase;
                } else if (randomChoice <= 80) {
                    current = upperCase;
                } else {
                    current = number;
                }
                password += current;
            }
            password += '-';
        }
        password.pop_back();
        passwordStorage.push_back(password);
        password.clear();
    }

    std::sort(passwordStorage.begin(), passwordStorage.end());
    passwordStorage.erase(std::unique(passwordStorage.begin(),
                                      passwordStorage.end()),
                          passwordStorage.end());

    for (std::string currentPassword: passwordStorage) {
        std::cout << currentPassword << '\n';
    }

    return 0;
}
```

Listing 5.5: Output of the *Prime Benchmark* Application.

```
Passes: 1033, Time: 5.000021;1;algorithm=base,faithful=yes,bits=1
```

Table 5.6: Runtime Evaluation for the *Prime Benchmark* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
i000002	CustomCompoundStmt	4.063 ms	0.08 %	0.08 %	1033
2418305	CXXMemberCallExpr	499.446 ms	99.59 %	99.59 %	1033
i000004	CustomCompoundStmt	0.126 ms	< 0.01 %	< 0.01 %	1033
2420840	IfStmt	< 0.001 ms	< 0.01 %	< 0.01 %	1033
Overhead		21.125 ms	0.42 %	0.42 %	8266
Scope		5014.148 ms			
Runtime		5014.148 ms			

5.1.4 Prime Benchmark

Finally, we want to examine an application that calculates as many prime numbers as possible within a period of 5 s. For this purpose, we will use the code that is freely available in a GitHub repository [20]. The calculation of prime numbers is often used to benchmark the performance of a processor. The *ROIProfiler* is used to find out which part of the software takes the longest time.

The result generated by the *Prime Benchmark* application can be seen in Listing 5.5. It can be seen that 1033 prime numbers were calculated in this run and that the runtime was approximately 5 s. The generated runtime statistic of the *ROIProfiler* can be seen in Table 5.6. It is clearly visible that the output of the data takes only a fraction of the calculations.

The next step is to find out which part of the calculation is the most demanding on resources. Therefore, the *ROIProfiler* is executed until it reaches the content of the function `setFlagsFalse(size_t n, size_t skip)`. The results generated by the *Prime Benchmark* application are shown in Listing 5.6. This time 1020 prime numbers could be calculated in a time span of 5 s, thirteen less than in the last run. The output generated by the *ROIProfiler* is shown in Table 5.7. Compared to the last run, not only 8266 but 685223 time values were calculated, which almost doubled the time used by the counters. In the statistics added by the *ROIProfiler* it can be seen that the content of the `while`-loop run takes 98.87 % of the total runtime, and thus the most resource intensive part of the application could be found. However, since there are only *leaf nodes* in the `while`-loop and these are combined into a `CustomCompoundStmt`, there would be no added value for further analysis. Furthermore, at the next stage, the point is reached where more counters are inserted than instructions that are to be measured, causing performance to drop drastically. We will discuss this behaviour in more detail in the overhead analysis of the *Prime Benchmark* program.

Listing 5.6: Output for the `CXXMemberCallExpr` of the *Prime Benchmark* Application.

```
Passes: 1020, Time: 5.004809;1;algorithm=base,faithful=yes,bits=1
```

Table 5.7: Runtime Evaluation for the `CXXMemberCallExpr` of the *Prime Benchmark* Application.

Identifier	ClassType	Runtime	Scope %	Total %	Calls
i000002	CustomCompoundStmt	13.682 ms	0.27 %	0.27 %	171360
2420840	WhileStmt	4922.025 ms	98.87 %	98.87 %	171360
Overhead		42.717 ms	0.86 %	0.86 %	685442
Scope		4978.425 ms			
Runtime		5018.688 ms			

5.2 Overhead Analysis

In order to determine how many additional resources are consumed by the inserted measurement code, a comprehensive analysis of the runtimes must be carried out. To achieve this, we will run the already known programs *Varying Loop Runtimes* 5.1, *Fibonacci Sequence* 5.3, *Password Generator* 5.4, and *Prime Benchmark* [20] in different environments and compare the runtimes. Furthermore, the *Password Generator* program is to be adapted in such a way that this time a number of the passwords to be generated is randomly selected. We will call this application the *Variable Password Size Generator*. Thus, a more precise analysis of the time added by the measurements is possible, since a different number of measured values is recorded in each run.

5.2.1 Method

All mentioned programs are compiled once without the added performance counter and once after they have been added by the *ROIProfiler*. The total runtime of both variants is measured using the *gnu-time* library [21]. By measuring the total runtime of the processes, it is possible to determine how much more time is required by all the changes made by the *ROIProfiler*. The data obtained from this measurement is primarily used to highlight the performance of the calculations and the output. In a second step, the times are measured directly in the *C++* applications using the *chrono* library. This means that only the pure running times of the instructions can be compared. For this purpose, one timestamp is saved when entering the main function and another one before the definition of the `print()` function. This eliminates the overhead that occurs when creating the process as well as the resource consumption of the `print()` function. This is especially important because the calculations and the output of the information add more time to the total runtime, but have no influence on the data needed for the profiling evaluation. The information obtained by measuring time with the *chrono* library gives

insight into the overhead caused by the added performance counters themselves. Each program is run a thousand times so that any outliers can be eliminated and an accurate analysis can be made.

During the analysis we will use the term *measurement value* when we talk about one measured time, which corresponds to exactly one added counter. A *measurement pair* consists of two *measurement values*, one recorded before the code block and one recorded after a code block. Further we will talk about the time needed to measure one whole code block. Measuring one code block may require more than one *measurement pair*, since code blocks in loops or recursive functions are measured more than once and the time of each iteration is accumulated. We will call this time the *total overhead per code block*.

The programs and scripts used for the analysis are freely available in a GitHub repository [6], allowing all test steps to be reproduced.

5.2.2 Hardware

In order to test the performance of the *ROIProfiler* on hardware of different levels of power, the evaluation is carried out on three systems. First, on a MacBook Pro Early 2015 with 8 GB 1867 MHz DDR3 memory and a 3.1 GHz dual-core Intel Core i7 processor running macOS Monterey version 12.1. We will call this setup *Test System 1 (MacBook)*. The second environment utilizes a 3,8 GHz AMD Ryzen 9 3900X dodeca-core processor with 96 GB 3200 MHz DDR4 memory, running ZorinOS 16 based on the Linux Kernel version 5.13.0. We will call this setup *Test System 2 (Ryzen)*. The last test device is a iMac 2021 with 8 GB 4266 MHz LPDDR-DDR4X memory and a 3,2 GHz Silicon M1 octa-core processor, running macOS Monterey version 12.1. We will call this setup *Test System 3 (iMac)*.

5.2.3 Varying Loop Runtimes

First, the runtime of the *Varying Loop Runtimes* application, which code we introduced in Listing 5.1, will be examined, focusing on the times measured with the *chrono* library, since *gnu-time* can only achieve an accuracy of 0.01 s. For this reason, only the inserted performance counters can be examined in this analysis and the `print()` function can be disregarded for the time being. The objective is to identify the difference in runtimes between the two program variants and to consider the time added per inserted performance counter.

The runtimes of each run performed on the AMD Ryzen test environment can be seen in Figure 5.1. The average of the durations of all runs without added performance counters is 16 300 µs, whereas the runtime after adding the performance counters is around 16 480 µs, which in result is a deviation of 180 µs. In this example, a total of four *measurement pairs* were recorded, whereby one measures the total running time and the other three the individual `for`-loops. Thus it can be calculated that each *measurement pair* consumed about 45 µs in this test environment.

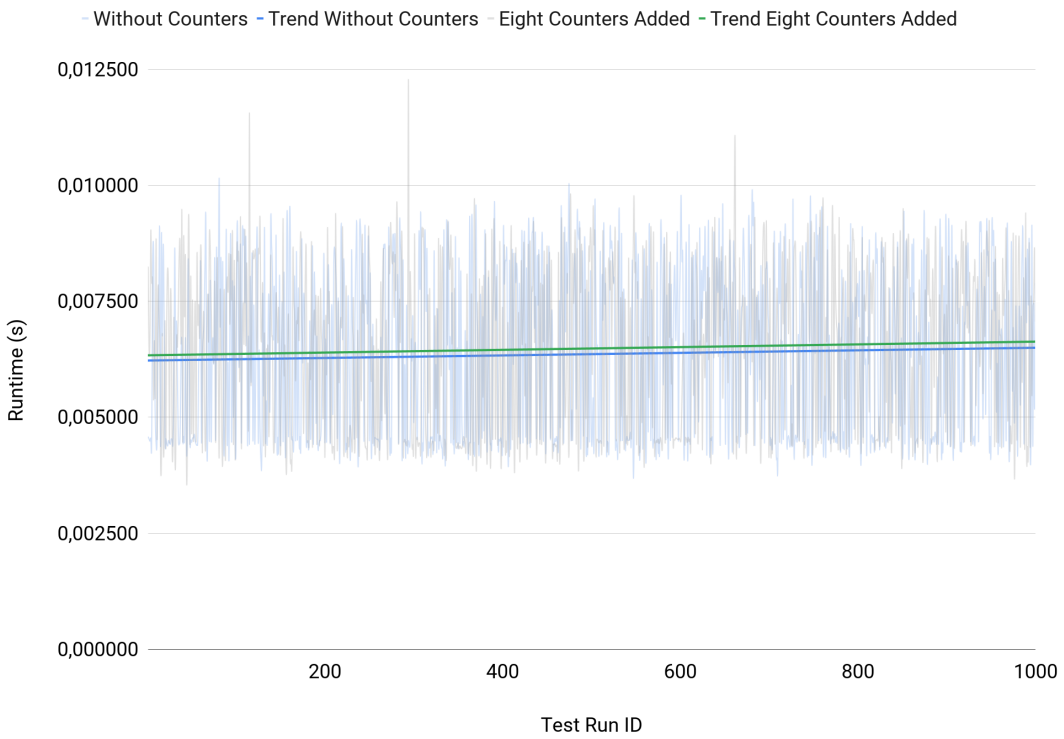

Figure 5.1: Runtime Comparison for the *Varying Loop Runtimes* Application. The diagram shows the runtime of each run. The version of the program without added counters is compared to the version with eight counters added. The trend lines indicate the average runtime. The test was performed on *Test System 2 (Ryzen)*.

Figure 5.2 shows the average durations per run on the three test environments, once with counter added and once without counter added. It can be seen that on average the programs with added counters consumed 0.34 %–2.28 % more time. If the increased time is calculated down to a single added performance counter, this results in a time of 20 µs for each *measurement value* taken across all environments.

It can be stated that especially with programs that have a short runtime and only a few counters have to be added by the *ROIProfiler*, only a marginal deviation can be recognized.

5.2.4 Fibonacci Sequence

The *Fibonacci Sequence* application that calculates the first thousand Fibonacci numbers will be looked at next. The code can be seen in Listing 5.3 In contrast to the previous application, we showed in Section 5.1.2 that two steps were needed to find the most resource-intensive part of the program.

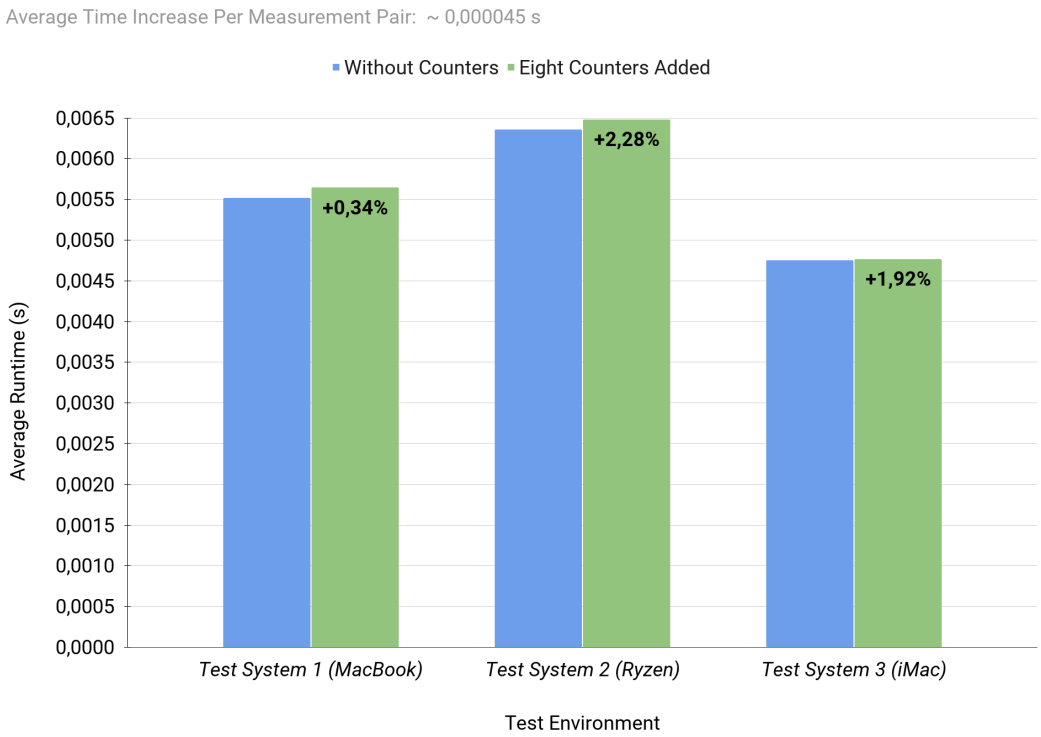

Figure 5.2: Runtime Deviation for the Execution of the *Varying Loop Runtimes* Application. The diagram shows the increase in runtime in percent when eight counters are added. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

In the first step, the total running time and the duration of two single instructions were measured, which amounts to a total of three *measurement pairs* or six *measurement values* recorded. In the second step, in addition to the total runtime, the runtime of the scope and two single instructions within a `for`-loop were calculated. Since the `for`-loop is executed exactly one thousand times and two instructions are measured within each, the total number of *measurement pairs* is two thousand plus two pairs for the total and scope runtime, which amounts to 2002 *measurement pairs* or 4004 *measurement values*. This example is intended to show that a distinction must be made between measurements that are performed only once and measurements that occur in a loop or a recursive function.

In Figure 5.3, the average runtimes of all environments for each run are given as trend lines. It can be seen that there is only a slight deviation between the variant without performance counter and the variant with six added *measurement values*. This difference amounts to an average of about 4.11 % on all test environments. In the variant with 4004 *measurement value* recorded, on the other hand, a more significant increase in runtime of

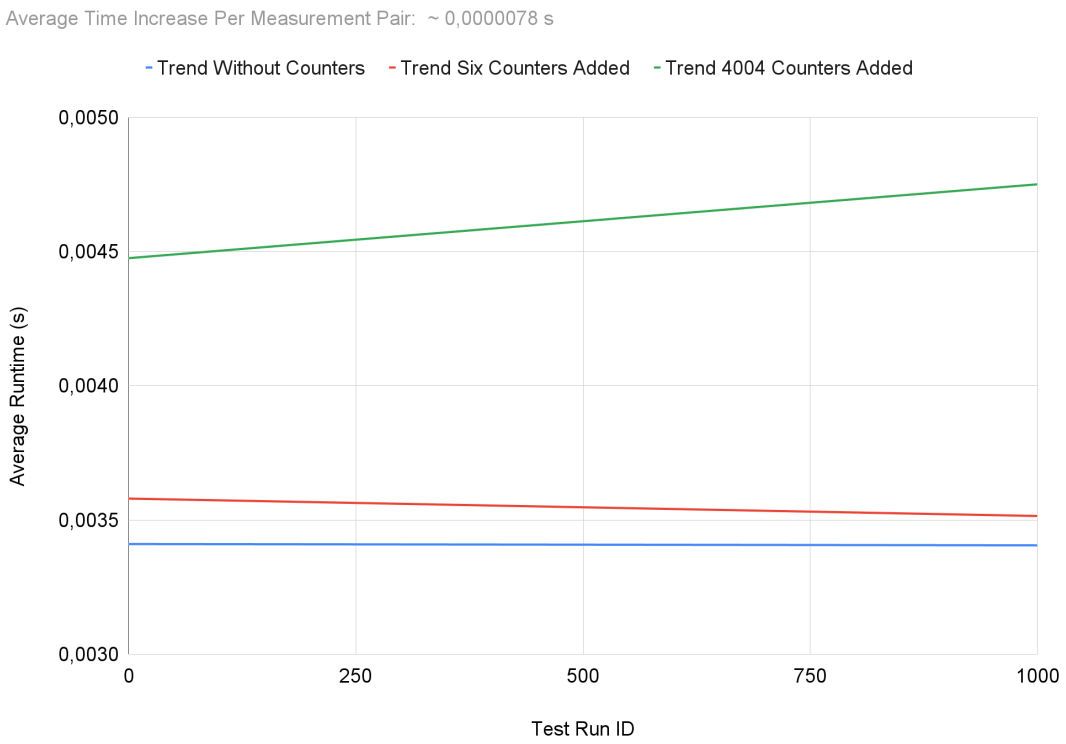

Figure 5.3: Runtime Comparison for the *Fibonacci Sequence* Application. The runtime of the program without counters is compared to the version with eight counters added and the version with 4004 counters added. The trend lines shown represent the average of the runtimes across all test environments. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

around 39.22 % can be seen. It can be stated that in this example there seems to be no clear dependence between the number of *measurement values* recorded and the added runtime, since the deviations do not increase proportionally to each other.

This behaviour can also be observed by looking at the average time increase per *measurement pair*. The program version without added counters takes on average 1136 µs, the variant with six counters added around 1182 µs and the variant with 4004 counters added around 1538 µs. If we now calculate the average added per *measurement pair* in this example, we get 7.8 µs per pair. However, it should be noted that a *measurement pair* costs around 15 µs in the first step and only 0.2 µs in the second, although significantly more *measurement values* were recorded. This can be explained by the fact that the memory does not have to be reallocated with each loop pass, but only the current value has to be overwritten.

Finally, the *total overhead per code block* can be considered. Although the costs of one

measurement pair is decreasing, the total time required to calculate the time of one code block is generally increasing. In the first step, the *total overhead per code block* cost around 15 µs, in the second step around 100 µs. This results from the fact that although the cost for a single *measurement value* is now lower, at the same time far more measurements are needed to calculate the time of one code block. Furthermore, it can also be stated that the execution time of the program is still very short and side effects caused by the hardware, the operating system or the time calculations of the runs can have a large impact on the final results. Consequently, the focus should shift to programs with a longer duration. Thus, we can take a closer look at the impact of the *measurement values* recorded on programs with a longer runtime.

5.2.5 Password Generator

The *Password Generator* application, which code can be seen in Listing 5.4, is to be examined next. We showed in Section 5.1.3 that a total of four steps were necessary in the profiling process to find the desired position in the code. In the first pass, eight *measurement values* were recorded for the time calculation of three code blocks, in the second eight million for the calculation of four code blocks, in the third fourteen million for calculating the time of two code blocks and in the last step 114 million for the calculation of three code blocks. The objective is to determine how the added performance counters affect programs with a longer runtime.

The average runtimes for each run across all test environments is shown in Figure 5.4. We have already discussed the deviations caused by a few added performance counters, so the first step is not considered for the following examples. The second step shows an increase in runtime of 6.11 %, the third step an increase of 10.68 % and the last step an increase of 85.64 %. It can be noted that the increase scales approximately with the number of *measurement values* recorded.

To view this effect from a different perspective, the average of the added time per *measurement pair* can be calculated. The result is 0.096 µs per *measurement pair* in step two, 0.1 µs in step three and 0.092 µs in step four, which amounts to an average of 0.097 µs. It can be seen that the runtime added per *measurement pair* remains at approximately the same level or even decreases although more *measurement values* are taken in total. This can be explained by the fact that the allocation of memory for the individual data structure entries no longer has such a large influence on the result. In the previous example it could be seen that it has a big influence whether three or four *measurement pairs* have to be allocated. However, in this application the deviation between the different numbers of *measurement pairs* is negligible.

In contrast, we want to look again at the time that must be spent to calculate the time of one code block. In the second step 0.064 s were spent, in the third step 0.18 s and in the last step 1.05 s. It can be seen that the *total overhead per code block* increases, since more calculations are needed to calculate the runtime of one code block. To investigate this further, the *Password Generator* program will be modified so that a different number of passwords is added in each run.

Figure 5.4: Runtime Comparison for the *Password Generator* Application. The figure shows the average runtime over all environments for each run. Program versions with a varying amount of added counters are compared to each other. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

5.2.6 Variable Password Size Generator

For the *Variable Password Size Generator* Application the fourth step of the *Password Generator* is to be adapted so that there is also a number of five data structure entries. This has the advantage that the difference caused by different numbers of data structure entries can be completely eliminated, since all runs have the same number of code blocks to be measured. Furthermore, the number of passwords should be randomly generated in the range from zero to two million so that any side effects caused by the hardware or the operating system can be distributed across all runs. Furthermore, the runs are now performed two thousand times to get a wider range of information.

Figure 5.5 shows the runtimes once with and once without added counters on the AMD Ryzen environment. In addition to the distinction between the program variants, a differentiation is now also made between the runtime of the counter itself and the runtime of the `print()` function. On average, for five measured code blocks, the runtime of the `print()` function is 0.04 s per run. Since in this example only a variable number

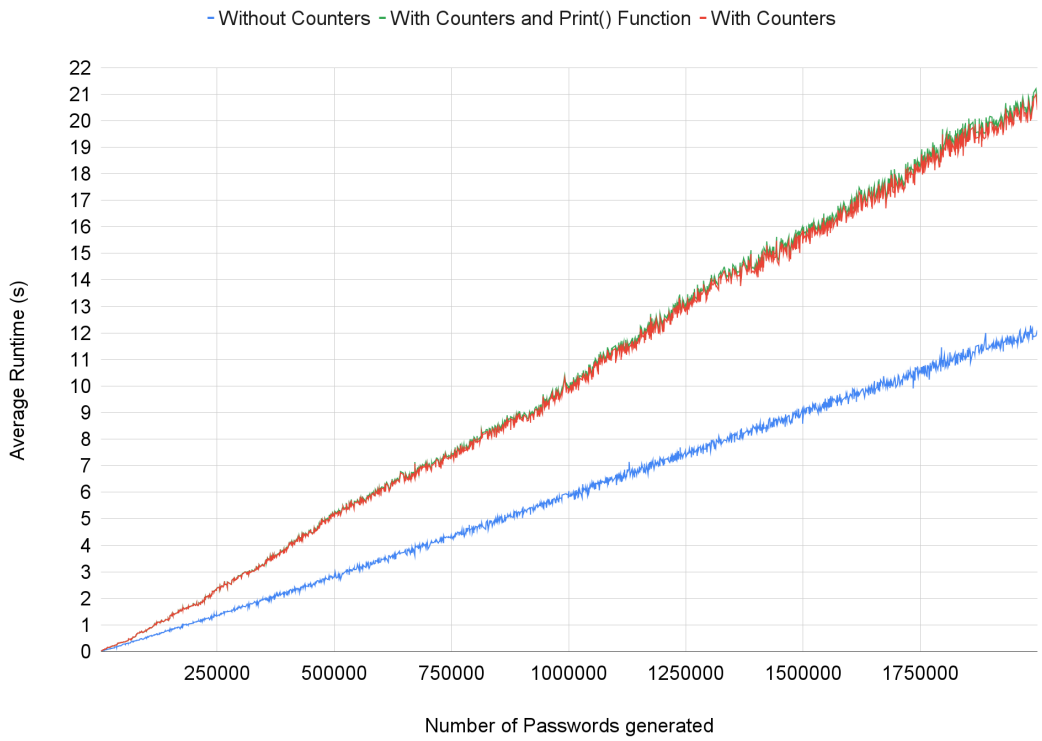

Figure 5.5: Runtime Comparison for the *Variable Password Size Generator* Application. The figure shows the increase of the runtime in relation to the generated passwords. The program version without added counters is compared to the version with counters and to the version with counters and `print()` function. The test was performed on *Test System 2 (Ryzen)*.

measurement values are recorded, but the number of code blocks which runtime statistic is to be printed remains the same, the runtime of the `print()` function does not change, but add a constant value to the total runtime. This is the case because the calculations of the runtimes of a loop are already made within the `startEvent()` function, thus an enormous amount of resources can be saved since two separate timestamps do not have to be stored for each loop run. It can be noted that the runtime of the `print()` function can only be increased by adding more code blocks to be measured, but adding single *measurement values* has no influence.

Furthermore, it should be calculated for all runs over all environments how much runtime is allocated to one *measurement value*. For this purpose, the total runtime can be divided by the number of performance counters used in this run. For the data of the program variant without inserted counters, the number of counters that would have been consumed can be calculated by scaling the number of passwords. An average value of 0.06 µs per

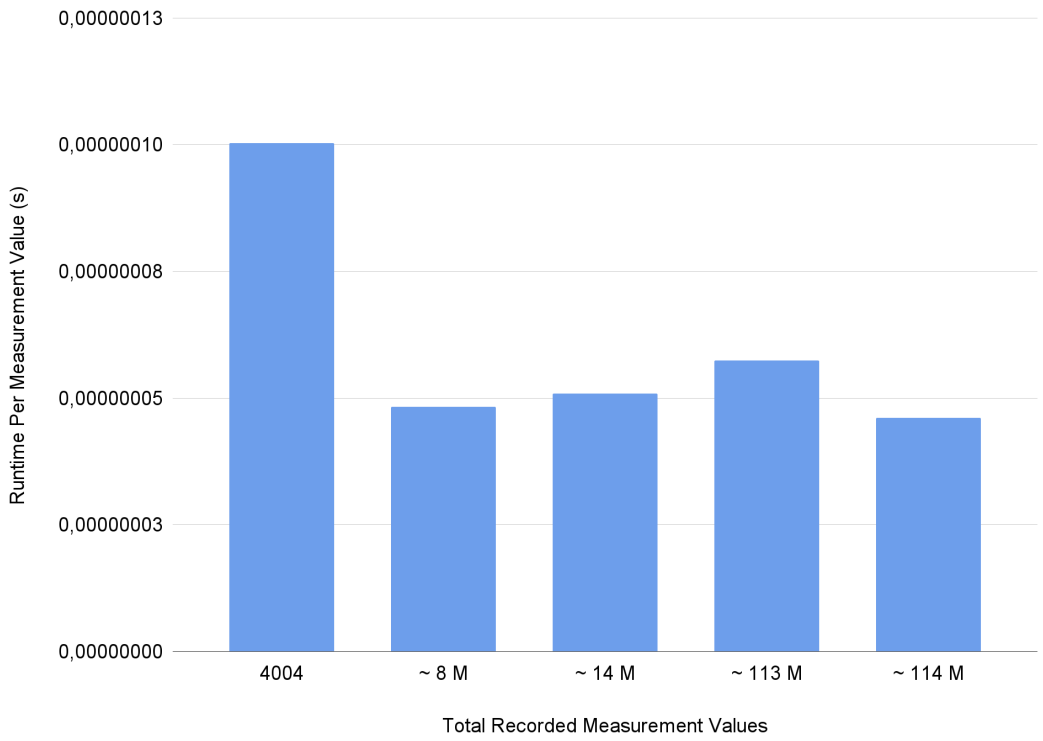

Figure 5.6: Time Per *Measurement Value* Comparison for Various Test Cases. The diagram shows the average time taken to record a *measurement value*. The *measurement values* calculated in the previous examples are compared to each other. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

measurement value can be calculated over the six thousand collected measurements. It can therefore be stated that a *measurement pair* in this example costs around 0.11 µs of running time. Comparing this value with those of the previous examples, especially the second step in the *Fibonacci Sequence* program and step two to four in the *Password Generator* application, it is noticeable that the deviation from the values calculated of these examples is very small. However, an average of 1.3 s was now spent calculating one code block. In the following we will compare the results of all programs in detail.

5.2.7 Time Per Measurement And Total Overhead Per Code Block

All values found for the runtime generated by one *measurement value* and the *total overhead per code block* of the various programs should be examined. Figure 5.6 lists the additional times required per *measurement value*. It can be seen that this value stabilizes as more times are measured. The sum of the five considered values is 0.06 µs. Figure 5.7 shows the *total overhead per code block* for all examples. These times increase as more

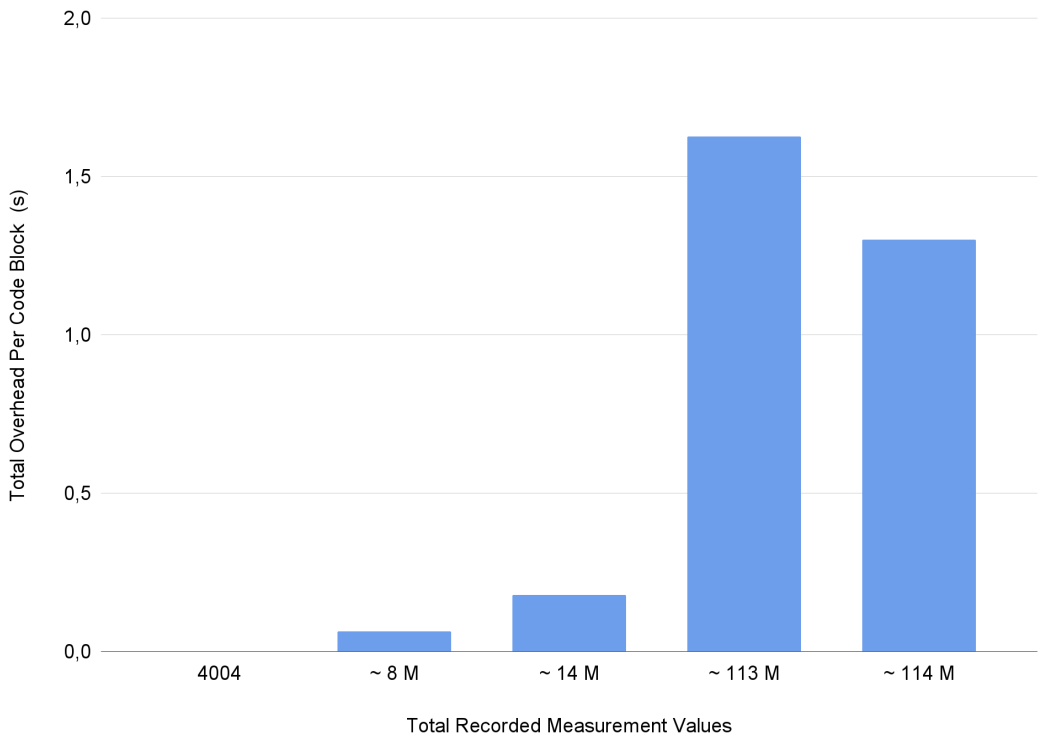

Figure 5.7: *Total Overhead Per Code Block* Comparison for Various Test Cases. The diagram shows how much overhead was generated by the measurement of one code block. The *total overhead per code block* calculated in the previous examples are compared to each other. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

measurement values are needed to calculate one code block. For the value calculated per *measurement value*, it can be said that it will not decrease further, but the *total overhead per code block* may greatly increase if more timestamps are measured and calculated.

In conclusion, it can be stated that the performance of the tool depends on three parameters. The first one is the number of code blocks to be measured, which can be neglected especially if many measurements are carried out. The second parameter is the *total overhead per code block*. This value rise when more readings are needed to measure a single code block. The last parameter is the time taken by *measurement value*. It was found that this value stabilizes with a large number of measurements and drops below 0.1 µs after about a thousand measurements have been taken. It should also be noted that although the execution time of the program increases by inserting further measurements, the quality of the profiling process is not necessarily affected. On one hand, the *ROIProfiler* recognizes about half of the additional time consumed, even

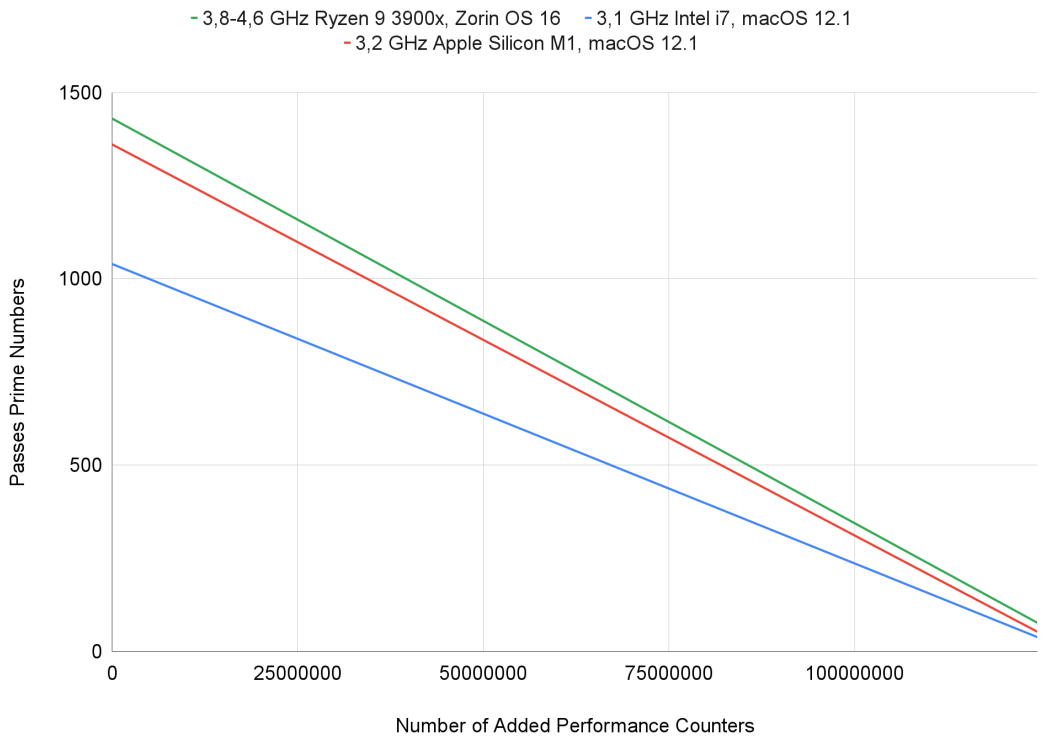

Figure 5.8: Prime Number Comparison for the *Prime Benchmark* Application. The diagram shows the ratio of calculated prime numbers to added performance counters. The runtime of all environments is compared to each other. The test was performed on *Test System 1 (MacBook)*, *Test System 2 (Ryzen)*, and *Test System 3 (iMac)*.

without further prediction models applied. On the other hand, the proportion of time consumed on one level enlarges evenly for all instructions, which means that statements can still be made about the most resource-intensive part of an application.

5.2.8 Prime Benchmark

After all statements about the added time of the *ROIProfiler* have been made, the *Prime Benchmark* [20] program will now be considered. In this example, no runtimes were measured, but rather calculated prime numbers. We showed in Section 5.1.4 that in order to reach the most resource-intensive part of the application, a total of six steps were required, whereby a different variable number of counters were inserted in each run, as this was dependent on the performance of the current run. In the first step, four *measurement values* were recorded, in the second, an average of 10241, in the third, a mean of 5105, in the fourth, approximately 1.2 million, in the fifth, an average of 0.8 million, and in the last step, an mean of 91 million. This test is particularly important

to show that the *ROIProfiler* can also be used for any third-party application.

Figure 5.8 shows the results of the test on all environments, with a linear regression trend calculated across all steps. Since the first five readings are very close to each other, the values of these runs have to be looked at separately, but in general it can be seen that fewer prime numbers could be calculated by adding the counters. In the range from zero to ten thousand inserted counters, an average of 1280 prime numbers were calculated. In steps four and five, around 1260 primes were calculated in a range of 0.8–1.2 million inserted measurements. In the last step, with 91 million counters, only 56 prime numbers could be calculated. This can be explained by the fact that the time for the calculation of the prime numbers was not changed, but more counters were inserted, thus the time for the actual calculation approaches zero.

For this specific application, it can be stated that the first five steps can be carried out without any problems, and only a slight performance drop can be detected. In the last step the calculation of the measured values takes more time than the calculations of the prime numbers themselves, whereby the performance drops enormously. Since a fixed runtime is used in this example, an infinite number of counters cannot be inserted, because at some point the profiling will take more time than the actual calculation of the prime numbers. However, we still succeeded in finding the most resource-intensive part of the program in Section 5.1.4, since we were able to determine which part of the calculation takes the longest time, even when calculating only a few prime numbers.

CHAPTER 6

Conclusion

In this thesis, an application was presented that enables *regions of interest* in single-threaded *C++* programs to be automatically wrapped with performance counters to find performance bottlenecks in the source code. This was achieved by first developing a concept for traversing the *Clang abstract syntax tree* and categorizing individual nodes. Afterwards, the functionality of the program was determined and a recursive model for the hierarchical traversal of the levels was found. Having worked out all the basic concepts, we took advantage of the *Clang* infrastructure by using the *LibTooling* library to develop a *Clang* tool that fulfilled our requirements. When developing this tool, we looked at how the concepts could be used to insert performance counters in the correct places in the source code. The developed program could then be used to test four applications of different complexities and runtimes for performance bottlenecks. This showed the workflow of the profiling process with this tool and proved that a resource-heavy instruction could be found in each program. Furthermore, we were able to perform a detailed overhead analysis by running each variation of the applications a number of times. In the process, we found out how much time is added by a counter and which parameters have to be taken into account when using the tool.

The implementation of the tool at this stage is limited by several constraints that were not addressed in the research question. First, it must be mentioned that the tool was developed to profile *C++* applications, but not *C* or *Objective-C*. *Clang* also offers infrastructures for traversing other languages from the *C* programming language family and thus the tool could be adapted based on the concepts created. To include other members of the *C* language family, for instance, an additional parameter could be created for the tool, with which the user can specify which programming language is to be transformed. The file extension could also be used to determine which programming language is involved. Depending upon the language in which the input program is written, further case distinctions could be added subsequently, which address concrete characteristics of the different languages. As an example the `CXXMemberCallExpr` can be considered. This

class exists in the object-oriented language *C++*, but it is not found in an *abstract syntax tree* of the *C* language, which means that a case distinction would be necessary.

Another limitation we have decided in the course of the work is that not all classes that are defined in the *Clang abstract syntax tree* are recognized separately by our tool. In this work, we have chosen a number of classes with which a variety of applications can be transformed. Further classes can be added however as desired to the present implementation. To achieve this it is necessary to consider other classes of the *abstract syntax tree* provided by *Clang* and determine whether they are *leaf nodes* or *parent nodes*. The existing case distinctions in the current implementation can easily be extended with new class queries.

It should also be noted that the tool has only been applied and tested on single-threaded applications. The concepts developed can also be used for multi-threaded applications, but further special cases must be considered in more detail. Due to the concurrent processing of the instructions, it must be ensured that the data structure that manages the runtimes is synchronized, or that there is a separate storage for each thread, which is reassembled at the end of the application.

Finally, it should be mentioned that the overhead analysis that is passed on to the user in addition to the statistics was not part of the research objective. Nevertheless, we succeeded in correctly allocating about half of the overhead without the implementation of additional calculations or predictions about the running times of the counters added. In order for the tool to be able to filter out the generated overhead even better, an algorithm could be designed, for instance, that determines how much overhead is generated in total based on the already correctly assigned share of overhead and on historical data. Thus, it would be possible to determine even more precisely how much runtime each code block really consumed.

Bibliography

[1] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, p. 120–126, jun 1982.

[2] GNU, "GCC, the GNU compiler collection - GNU project." `https://www.gnu.org/software/gcc/`. [accessed: 25-January-2022].

[3] GNU, "GNU gprof." `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`. [accessed: 8-February-2022].

[4] J. Thiel, "An overview of software performance analysis tools and techniques : From gprof to dtrace," *Washington University in St. Louis, Tech. Rep*, 2006.

[5] Clang, "Clang C language family frontend for LLVM." `https://clang.llvm.org/`. [accessed: 18-February-2022].

[6] M. Hagn, "Github - maxhagn/roiprofilercpp." `https://github.com/maxhagn/ROIProfilerCPP`. [accessed 6-May-2022].

[7] Clang, "Libtooling — Clang 15.0.0git documentation." `https://clang.llvm.org/docs/LibTooling.html`. [accessed 18-February-2022].

[8] F. Schlebusch, Y. Müller, S. Wienke, J. Miller, and M. S. Müller, "Pint: Pattern instrumentation tool for analyzing and classifying HPC applications," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 71–80, 2018.

[9] A. Hück, J. Utke, and C. Bischof, "Source transformation of C++ codes for compatibility with operator overloading," *Procedia Computer Science*, vol. 80, pp. 1485–1496, 12 2016.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, (San Jose, CA, USA), pp. 75–88, Mar 2004.

[11] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[12] LLVM, "The LLVM compiler infrastructure project." https://llvm.org/. [accessed 10-February-2022].

[13] M. Klimek, "The Clang AST - a tutorial." https://www.youtube.com/watch?v=VqCkCDFLSsc. [accessed 16-January-2022].

[14] LLVM, "The LLVM compiler infrastructure project." https://llvm.org/devmtg/2013-04/#tuto1. [accessed 16-January-2022].

[15] Clang, "Introduction to the Clang AST — Clang 15.0.0git documentation." https://clang.llvm.org/docs/IntroductionToTheClangAST.html. [accessed 20-January-2022].

[16] Clang, "ClangCheck — Clang 15.0.0git documentation." https://clang.llvm.org/docs/ClangCheck.html. [accessed 12-February-2022].

[17] Clang, "Welcome to Clang's documentation! — Clang 15.0.0git documentation." https://clang.llvm.org/docs/index.html. [accessed 19-February-2022].

[18] Clang, "Choosing the right interface for your application — Clang 15.0.0git documentation." https://clang.llvm.org/docs/Tooling.html. [accessed 12-February-2022].

[19] CPP Reference, "Date and time utilities - cppreference.com." https://en.cppreference.com/w/cpp/chrono. [accessed 15-February-2022].

[20] Plummer's Software LLC, "Github - plummerssoftwarellc/primes: Prime number projects in C#/C++/Python." https://github.com/PlummersSoftwareLLC/Primes. [accessed 2-February-2022].

[21] Clang, "time(1) - linux manual page." https://man7.org/linux/man-pages/man1/time.1.html. [accessed 15-February-2022].