Analysis and Methods
for Supporting Generative Metaprogramming
in Large Scale C++ Projects

Doctoral dissertation
2014

József Mihalicza
jmihalicza@gmail.com

Thesis advisor: Dr. Zoltán Porkoláb, docent
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School
Doctoral program: Foundations and Methodologies of Informatics
Head of the doctoral school: Prof. Dr. András Benczúr
Head of the doctoral program: Prof. Dr. János Demetrovics
Copyright

Microsoft, Visual C++, Visual Studio and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.
# Contents

List of Figures vi

List of Tables viii

List of Algorithms x

Listings x

Acknowledgements xii

1 Introduction 1

1.1 Generative program systems ........................................ 1

1.1.1 DRY principle .................................................. 1

1.1.2 Generics ....................................................... 2

1.2 Large-scale program systems ....................................... 3

1.3 Addressed issues .................................................. 4

1.4 Thesis structure .................................................. 5

2 Analysis of unity build 7

2.1 Motivation ......................................................... 7

2.2 Root cause of slow C++ builds ................................... 8

2.3 Methods for reducing build times ................................ 9

2.4 Measurements ..................................................... 11

2.5 Drawbacks of unity build ......................................... 14

2.5.1 Increased compiler load ....................................... 14

2.5.2 There is no such as local ...................................... 15

2.5.3 Errors caused by automated unification ....................... 19

2.5.4 IDE wants to compile all implementation files ............. 19

2.6 Strict equivalence checks ......................................... 19

2.6.1 $EQ_u$ ......................................................... 20

2.6.2 $EQ_t$ ......................................................... 21

2.6.3 $EQ_a$ ......................................................... 22
2.7 Correctness and applicability ........................................... 25
2.8 Evaluation ................................................................. 27
2.9 Contribution .............................................................. 27

3 Generative metaprogramming ............................................. 29
  3.1 Preprocessor metaprogramming ........................................ 30
    3.1.1 Preprocessor patterns ........................................... 31
    3.1.2 Pitfalls of using the preprocessor .............................. 33
    3.1.3 Boost.Preprocessor ................................................ 34
    3.1.4 Evaluation .......................................................... 34
  3.2 Template metaprogramming in C++ .................................... 35
    3.2.1 Applications ....................................................... 35
    3.2.2 Evaluation .......................................................... 40

4 Template metaprogram instrumentation .................................. 42
  4.1 Instrumentation in general ............................................ 42
  4.2 Motivation for template metaprogram instrumentation .............. 45
  4.3 The idea ................................................................. 46
  4.4 Implementation ........................................................ 47
    4.4.1 Preprocessor ....................................................... 47
    4.4.2 #line filter ....................................................... 49
    4.4.3 Annotator .......................................................... 49
    4.4.4 Instrumentator .................................................... 52
    4.4.5 Compiler ............................................................ 54
    4.4.6 Warning parser .................................................... 54
    4.4.7 Position adjust .................................................... 55
  4.5 Limitations ............................................................. 57
    4.5.1 Pattern recognition difficulties ............................... 57
    4.5.2 Inheritance ........................................................ 58
    4.5.3 Template processing mechanism ............................... 60
    4.5.4 Compiler (in)dependence ....................................... 60
  4.6 Related work .......................................................... 61
  4.7 Contribution .......................................................... 62

5 Template metaprogram debugging ....................................... 63
  5.1 Motivation ............................................................... 63
  5.2 Debugging run time vs. TMP ......................................... 64
  5.3 Different ways to implement a template debugger .................. 68
    5.3.1 Instrumentation based approach ............................... 68
    5.3.2 Modifying the compiler ....................................... 69
  5.4 Applications .......................................................... 72
5.4.1 IDE extension ........................................... 72
5.4.2 Standalone IDE ........................................ 75
5.4.3 Templar .................................................. 75
5.5 Related work .............................................. 78
5.6 Contribution .............................................. 80

6 Template metaprogram profiling ................................... 81
6.1 Motivation .................................................. 81
6.2 Different approaches of implementing a TMP profiler ............. 82
  6.2.1 Measure full compilation time (FC) .......................... 83
  6.2.2 Warning instrumentation and out-of-process measurement (WO) ............................................. 83
  6.2.3 Warning instrumentation and in-process measurement (WI) ............................................. 85
  6.2.4 Built-in support for profiling (BI) .......................... 85
6.3 Analysis ..................................................... 86
  6.3.1 Example codes used in measurements .......................... 86
  6.3.2 Measuring preprocessing overhead of FC ..................... 88
  6.3.3 Using BI for the inheritance/reference comparison .......................... 89
  6.3.4 Measuring instrumentation overhead .......................... 91
  6.3.5 Differences between debug and release builds .................. 95
6.4 Conclusion ................................................ 97
6.5 Related work .............................................. 98
6.6 Contribution .............................................. 100

7 Type-preserving heap profiler for C++ ................................ 101
7.1 Motivation ................................................ 101
7.2 Requirements ............................................. 103
7.3 System design ............................................ 103
  7.3.1 System structure ........................................ 103
  7.3.2 Data structures ........................................ 104
7.4 Implementation details ..................................... 106
  7.4.1 Capturing memory operations .................................. 106
  7.4.2 Retrieving type information .................................. 107
  7.4.3 Pitfalls ............................................ 110
  7.4.4 Call stack ............................................ 115
7.5 Usage ..................................................... 117
7.6 Visualisation ............................................. 123
  7.6.1 Algorithmic challenge behind timeline view .................. 125
7.7 Usage examples ........................................... 134
7.8 Related work ............................................. 136
7.9 Contribution ............................................. 138
List of Figures

2.1 Flowchart of C++ compilation. ................................. 8
2.2 Size of the #include section compared to the actual code section before and after preprocessing in OpenSceneGraph. .......... 9
2.3 Flowchart of C++ compilation with unity build. ................. 12
2.4 Multiple processing of the same header files are eliminated by unity build. ................................................. 12
2.5 Loosing independence and locality in unity builds. .......... 15
2.6 Unity build as an option. ........................................... 18
2.7 Recursive uniting. .................................................. 20

3.1 Standard code transformation steps in C++. ...................... 29
3.2 The \texttt{#include} macropath pattern. .......................... 31
3.3 An extra level of indirection to workaround the comma problem in macros. .................................................. 33

4.1 C++ template metaprogram instrumentation workflow. ............ 46
4.2 Templight control flow. ............................................. 48
4.3 Instantiation events of the code in Listing 4.9, base and variant versions. .................................................. 59

5.1 Sequence diagram of starting a TMP debugging session with the Templight VS add-in. ............................................. 73
5.2 Debugging the classic \texttt{Factorial} example with the TMP debugger add-in. .................................................. 74
5.3 Debugging the classic \texttt{Factorial} example with the standalone TMP debugger. .................................................. 76
5.4 Debugging the classic \texttt{Factorial} example with Templar. ....... 77
5.5 Breakpoints are defined as regular expressions in Templar. ....... 78

6.1 Workflows of the FC, WO, WI and BI TMP profiling approaches. 84
6.2 Duration of full compilation versus preprocessing separately for the units of Bayes++. ............................................. 88
6.3 Measurement results of the BI method. ............................. 91
6.4 Instrumentation overhead. .......................................... 92
6.5 Instrumentation overhead for low and high values. ................ 92
6.6 Instrumentation overhead when warnings are enabled. ............. 93
6.7 Time spent in outputting warnings in the WI method. .............. 94
6.8 Raw times and corrected times in the \texttt{measure} version when using the WI method. ............................................. 94
6.9 Normal, instrumented and corrected times of Bayes++ compilation using the WI method, summarised by compilation units. ....... 95
6.10 Total times and corrected times of the 200 most time consuming template instances of Bayes++ compilation using the WI method. ............................................. 95
6.11 Time spent in templates when compiling different units of Bayes++, measured with the BI method. ......................... 96
6.12 Distribution of template instance multiplicity among Bayes++ units. ............................................. 96
6.13 The two views of \texttt{ProfileDataViewer}. .......................... 99

7.1 Structure of my profiler framework. .................................. 104
7.2 Example of the typical memory layout of an array with non-trivial
destructor. ................................................................. 111
7.3 The connection between allocation entries, call stack and CCF. . . . 118
7.4 A complete filter graph. .................................................. 124
7.5 Memory view in Memory Usage Analyser displaying pool allocation
entries and the objects in the pools. .................................... 125
7.6 Representation of a subset as a vector. .................................. 130
7.7 Timeline view of loading a text file in Notepad++. ......................... 135
7.8 Timeline view of loading a text file in Notepad++ after optimisation. 136
A.1 Generic matrix implementation in C. ..................................... 175
B.1 Templight grammar to detect template constructs. ......................... 182
B.1 (continued) Templight grammar to detect template constructs. ... 183
B.2 Templight control flow with the listings of the Fibonacci example. . 186

List of Tables

1.1 Pros and cons of using extra build steps for code generation or using
host language constructs. ................................................. 2
1.2 Chapters and their content. .............................................. 6
2.1 Preprocessing overhead without and with unity build. .................... 13
2.2 Full build times in seconds without and with unity build. ............... 13
2.3 Normalised compilation speeds. ......................................... 13
2.4 Amount of changes needed to make the libraries unity build capable. 18
3.1 C preprocessor language constructs ..................................... 30
3.2 Preprocessor level data types. .......................................... 34
4.1 Types of trace events and their properties. ................................ 54
4.2 Warning trigger codes and the generated warnings with g++ 4.8.2
and clang 3.1. ............................................................. 61
5.1 Relation between presumed and real execution in different debugging
scenarios. .................................................................... 64
5.2 How specific language elements appear in run time functional and
C++ template metaprograms. ................................. 65
5.3 How specific run time execution elements relate to C++ template
metaprogram execution. ...................................... 66
5.4 How specific run time execution elements relate to C++ template
metaprogram execution. ...................................... 66
5.5 Source files containing template processor codes in gcc and clang. 70

6.1 Hook points when applying the BI method. ....................... 90
6.2 Implementation requirements and precision of different TMP pro-
fil ing methods. ................................................. 97

7.1 Signatures of enter and exit functions in different compilers. 116

8.1 Independent citations of my research. .......................... 142
8.2 Fellow citations of my research. ................................ 143
8.3 Applications of and references to my research. ................ 144

D.1 Real usage statistics of timeline view, partial views. .......... 200
D.2 Real usage statistics of timeline view, full views, \( d < 1000 \). 201
D.3 Real usage statistics of timeline view, full views, \( d \geq 1000 \). 202
List of Algorithms

7.1 Unoptimised version of time slice processing. .................. 127
7.2 Optimised time slice calculation. ................................. 128
7.3 Optimised time slice calculation using the subset and iterator types. 129
7.4 Implementation of subset and iterator types. .................. 131
7.5 Visualising $U(M)$ in $O(n_a)$ time after preparation. ........ 133

Listings

3.1 Workaround for macro arguments that may contain commas. .... 33
4.2 Annotation result of the code in Listing 4.1. ...................... 50
4.1 Example code for demonstrating template class annotation. ..... 51
4.3 Sample code before instrumentation. .............................. 52
4.4 Sample code after instrumentation. ............................... 52
4.5 The beginning of a linemap file. ................................ 53
4.6 Sample trace file fragment. ........................................ 55
4.7 Context sensitivity in the C++ grammar. .......................... 58
4.8 Template function declaration containing ) token in the argument list. ................................................. 58
4.9 Two different definitions of the Factorial metafunction. ....... 59
6.1 Test code for comparing efficiency of using inheritance or plain reference to create recursion in a simple template metaprogram. .... 87
6.2 Output of run #1 (no print version) when applying the BI method. 90
6.3 Output of run #2 (print version) when applying the BI method. .. 91
6.4 How timestamps appear in the entries of a Templight trace file. .. 99
7.1 Implementation of the NewTrick class. ............................ 108
7.2 Type identifiers in the map file. ........................................ 109
7.3 Obtaining pointer to allocated area in array allocations. ........... 112
7.4 Text file listing allocation entries. ..................................... 120
7.5 Code producing double delete. ......................................... 121
7.6 CCF dump. .................................................................. 121
7.7 Muar file example. ......................................................... 122
A.1 Using the generix matrix implementation. ............................ 174
A.2 Example application of the \texttt{X} macro and the direct product patterns. 176
A.3 Using the Boost.Preprocessor framework to implement compile time matrix multiplication. .................................................. 177
A.4 Using preprocessor time matrix multiplication. ....................... 178
A.5 Preprocessor output of the matrix multiplication usage example. 178
A.6 Type list with an indexing metafunction. .............................. 179
A.7 Template class with 5 integer parameters is actually a fixed 5 length compile time array of integers. .................................... 180
A.8 Usage examples of Boost.Bind. ......................................... 180
A.9 CRTP used for implementing instance counting as a mixin. ......... 181
B.1 The \texttt{spirit} grammar used in Templight. .......................... 184
B.2 Original source code of the Fibonacci example. ...................... 187
B.3 Preprocessed source code of the Fibonacci example with \texttt{#line} directives. ......................................................... 187
B.4 File/line mapping of the Fibonacci example. ........................... 188
B.5 Unaltered preprocessed source of the Fibonacci example without \texttt{#line} directives. ......................................................... 188
B.6 Annotation of the Fibonacci example. .................................. 189
B.7 Line mapping of the Fibonacci example. ............................... 190
B.8 Instrumented code of the Fibonacci example. .......................... 190
B.9 Compilation messages of the Fibonacci example. ..................... 191
B.10 Trace file of the Fibonacci example before position adjust. ..... 195
B.11 Position correct trace file of the Fibonacci example. ............... 196
C.1 Trace file with profile information and memoisation events generated by a modified version of the clang compiler. ............... 198
Acknowledgements

I owe a lot to my family: my wife, Szandra, and my daughter, Kinga, for respecting my PhD ambitions and letting me immerse deeply in my research. I would like to thank Zoltán Porkoláb, my thesis advisor, the continuous support, all the consultations and the valuable discussions. Without his professional approach to teaching C++ and the way he oriented students to interesting challenges I definitely would not step onto the way of becoming a doctor. Many fellow students and (ex-)colleagues contributed to my works either by being coauthors, or via fruitful arguments. I do not even try to enumerate them all. Those who added the most to the topics of this dissertation are Zoltán Borók-Nagy, Ábel Gábor, Viktor Májer, Norbert Pataki, Ábel Sinkovics and Ádám Sipos.
Chapter 1

Introduction

1.1 Generative program systems

Generative programming is an approach in which carefully generalised software modules are created and then concrete variations of these generalised modules are combined to fulfil the specific needs in different situations [16, 17]. A program system is generative if it employs generative programming. The transition from the generalised version to the concrete variation can either be run time, load time, link time, compile time, or some mixture of these. This thesis discussion is restricted to the compile time version, where parts of the program code are generated by the production toolchain. This code generation step can be explicit, when the build environment performs extra transformation step(s) before compilation. Some languages contain code transformation constructs, such as simple text replacement tools (macros) or even code generators at a more semantic level (generics, templates). With these built-in tools a generative program can be written in a single programming language, making the multiphase compilation model invisible to the build environment. There are pros and cons of both approaches. Table 1.1 mentions a few of these. What such a code generator is good for? Among its most frequent usages are applying the DRY principle and generic constructs. Let us have a closer look on each of the two with some examples.

1.1.1 DRY principle

The DRY principle is short for “do not repeat yourself” [18]. The point here is consistency. Whenever the same information is present multiple times, it is an implicit task to keep them consistent, otherwise bugs occur. Even if they are placed close to each other and are clearly visible, like two counting loops with the same interval, it is an extra task that has to be kept in mind. A more dangerous case is when the repetition of some expression is not that clear. If the occurrences
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Aspect</th>
<th>explicit code generation step</th>
<th>using host language constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation primitives</td>
<td>proper transformation language can be used with many features</td>
<td>the set of possible transformation constructs is limited by the host language</td>
</tr>
<tr>
<td>Language semantics</td>
<td>external tools treat code as text</td>
<td>the compiler is aware of the role of the symbols in the code, and the transformation can take it into account</td>
</tr>
<tr>
<td>Intrusivity</td>
<td>a library forces its users to incorporate the transformation step into their toolchain</td>
<td>transformation steps are performed automatically for the client code as well</td>
</tr>
</tbody>
</table>

Table 1.1: Pros and cons of using extra build steps for code generation or using host language constructs.

does not look the same, or they are in separate files, only the original author is (less and less as time goes) likely to remember the consistency constraints. DRY not only saves you from having to apply a change consequently. In many cases, in particular what generative programming is mostly good for, a long, tedious, or even unfeasible process can be automated.

Note that DRY is not the ultimate way to go. As it is pointed out in [19], there are certain cases when violating the DRY principle is more beneficial than blindly obeying it.

1.1.2 Generics

Generics are the application of the DRY principle in a very direct form. They enable the programmer to write the same algorithm or data structure or big program components once in a general form, which later can be reused in concrete situations, substituting the free parameters with actual values. These free parameters are usually types or constants. Some languages also allow generic parametrisation with function references or other generic constructs. The Standard Template Library (STL) [20], of which interface is part of the C++ standard [21], demonstrates the power of such generic containers and algorithms. In the recent years more and more generic libraries have become available. Another notable example is Boost [22], one of the main sources of new components in the standard libraries of C++11 [21] and C++14 [23]. These are specific to C++ but other languages also
adopted generic constructs in different ways [24]: Ada, Java, C# [25], Eiffel [26], Scala [27] just to name a few.

1.2 Large-scale program systems

There is no exact definition for what large-scale means. It is obvious that a few thousand lines of code can not be treated as large-scale. But where is the transition? My definition is the following: a program system is large-scale, if it is not feasible for a single programmer to know each subtle detail of the implementation. When the source is of millions lines of code, this is certainly true. These systems usually already have some history, it takes years to put together such a big code base. The zlib compression library counts 13k LOC with 2 main authors [28]. Linux kernel 3.2 has 15M LOC with 1316 developers involved [29]. Daniel Chapman in [30] says: “The kernel project has long since grown to a size where no single developer could possibly inspect and select every patch unassisted.” Interestingly, complexity tends to grow together with size regardless of programming language, paradigm or methodology [31].

An important aspect is build time. Smaller systems usually build in a few minutes. It is not uncommon for a large-scale system to have build times of hours [32, 33]. Not only the proper implementation of a task gets more and more difficult as complexity grows: it can take hours to compile any version that has modifications in a widely used interface file [34]. Moreover, that modification will be updated to the working copies of each developer, resulting in further build times.

Build time is an example of non-functional requirements. Another category of non-functional requirements is run time performance, in terms of speed, responsiveness, CPU and/or memory load, usage of exclusive resources etc. Performance problems are typically sneaky. Performance indicators rarely get worse dramatically. It is rather common that they have a slow tendency of becoming worse and worse [35]. Furthermore, these indicators are not independent. If you have achieved an improvement in one performance type, your change may have caused another type to worsen [36].

Comprehension also gets more and more important as size grows. This need is present at each abstraction level: good naming convention for the source code elements, design review, communicating major architectural decisions. If communication is insufficient, diverse solutions appear to similar problems. This not only leads to an inconsistent design, but further extends the code base to maintain, with each its long term consequences listed in this section. If the system is big enough, the ability of effective searching is vital. This shouts for a unified language.
1.3 Addressed issues

We have seen that large-scale systems have numerous additional issues to handle compared to smaller ones. This is, of course, true for large-scale generative systems as well. My theses focus on specific areas. Instead of giving general answers to general questions, my goal was providing readily applicable methods, approaches to some selected every day industrial problems.

One key cut of scope is the programming language observed. My results have a strong bias towards C++. Though some of them can be generalised to be applicable to different environments, it was not a direct goal. This choice was not incidental. C++ is a language, where the language design facilitates multiparadigm programming [37]. Object oriented approaches can be used together with generative, functional and procedural parts. Also, support for system programming, good compatibility with the quasi-standard C ABIs [38] made it one of the most widely used language in the industry [39]. Consequently there are many real life examples of the topics covered.

My theses have three strong focuses: build time, diagnostics and analysis of C++ template metaprograms, and type-preserving analysis of heap memory usage in C++.

My first thesis analyses the application of unity build, which is a method to radically reduce build times of programs written in languages that use the C preprocessor as a replacement for native module support. I examine the pros and cons of applying unity build and show the results of efficiency measurements. I derive a set of criteria that allows automatic detection of unwanted unification side effects.

It was two decades ago when the first C++ template metaprogram was published [40, 41]. It turned out that template metaprogramming in C++ is actually an embedded functional language [42, 43, 44]. Moreover, it is Turing-complete [45, 46], so it can be used for solving, theoretically, any algorithmic problem. Generative constructs can be implemented in a type safe manner with templates. There are numerous libraries and applications using these techniques. TMP stands for these template meta programming techniques in the rest of the document. Still, after this many years and the wide usage, there is not a convenient toolset that supports TMP. My next three theses fill this gap, provide methods for TMP diagnostics and analysis. TMP instrumentation is an approach that makes analysis possible even if we cannot modify the compiler. TMP debugging shows how the standard debugging, like breakpoints, step in, out, can be interpreted and applied on template metaprograms. Profiling is about measuring performance properties, like running time or memory usage. In case of TMP a profiler should provide these information about template instances.

My fifth thesis presents a method that allows type information to be preserved during memory usage analysis of C++ programs. I go through the implementation
details with strong focus on the possible pitfalls and provide solutions for them. Then I describe our visualiser tool which is a convenient graphical interface for effective heap behaviour analysis. Displaying timeline views (see 7.6) in the visualiser tool efficiently even for large data sets is a non-trivial problem. As part of this fifth thesis chapter I analyse this specific algorithmic problem and show optimal algorithms.

1.4 Thesis structure

The introduction chapter (1) explains what generative program systems are, and describes what I treat large-scale in this thesis. This chapter gives the foundation of the presented results, defines their context. Then we have a quick overview of what main areas the theses address.

The first thesis chapter (2) discusses build time issues in general, then presents the results of case studies with unity build. Besides its pros, the drawbacks and practical applicability also get strong focus.

The next three of my five theses are related to C++ template metaprogramming. The generative metaprogramming chapter (3) introduces the main concepts of this area, which is important for understanding the achieved results and their role.

Then the three TMP related theses are presented. Instrumentation comes first in Chapter 4, as it is the base of the three TMP theses. I present an approach for extracting template processing details of the compilation process with or without modifying the compiler. There are cases when modifying the compiler is not an option. Even if the source code of the compiler is available, patching it may not be desirable for several reasons. It introduces risk and makes migration to another version of the compiler harder.

In case of run time programs (as opposed to metaprograms which run in compile time) we have rich sets of available debugging tools. This is not true for C++ template metaprograms. In Chapter 5 on TMP debugging we will see how the result of instrumentation can be used for applying the same debugging techniques on TMP’s as in the case of run time programs. Methods are presented that enables the developer using breakpoints, step in/out/next commands, examine instantiation stacks (the TMP equivalent of the run time call stacks).

Profiling is getting detailed reports about performance. In case of TMP this is mostly compile time and memory. The original purpose when designing the template construct of C++ was not supporting extensive metaprogramming, but only to provide a general enough generic construct. Consequently compiler writers did not necessarily treated thousands or even millions of template instantiations a typical case when making architectural design decisions about the implementation
of the template instantiation procedure. When the TMP paradigm appeared, it soon turned out that indeed, compilers do not scale very well if we start using certain template constructs heavily [47]. Being aware of what constructs to avoid with the compiler used is crucial for keeping build times low. A tool that pinpoints the problematic areas comes very handy in such a situation. My thesis presents approaches that make profiling possible for template metaprograms, thus fulfilling this need.

The last thesis chapter (7) turns the focus from compile time behaviour and resource usage to run time heap memory usage. There are many good tools for analyzing dynamic memory usage in C++ applications, but most of them captures the allocation events at such a point where the type information is already lost. My method brings this capture point earlier, where the type information is available, and extends the gathered analysis data with this type information.

The summary chapter (8) sums up my contributions and enumerates their impact both in academics and in the industry. Then I tell a few word on my actual research interests.

Those source code listings that are not crucial parts of the discussion have been moved out to the appendix residing after the bibliography.

<table>
<thead>
<tr>
<th>chapter</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define context and focus</td>
</tr>
<tr>
<td>2</td>
<td>Thesis 1: Analysis of unity build</td>
</tr>
<tr>
<td>3</td>
<td>Foundations for Theses 2, 3 and 4</td>
</tr>
<tr>
<td>4</td>
<td>Thesis 2: Template metaprogram instrumentation</td>
</tr>
<tr>
<td>5</td>
<td>Thesis 3: Methods for template metaprogram debugging</td>
</tr>
<tr>
<td>6</td>
<td>Thesis 4: Methods for template metaprogram profiling</td>
</tr>
<tr>
<td>7</td>
<td>Thesis 5: Type-preserving heap profiler</td>
</tr>
<tr>
<td>8</td>
<td>Recapitulation</td>
</tr>
<tr>
<td>A–E</td>
<td>Supplementary details</td>
</tr>
</tbody>
</table>

Table 1.2: Chapters and their content.
Chapter 2

Analysis of unity build

Unity build is a technique to make both full and incremental build times of C++ projects significantly lower. This chapter pinpoints the root cause of slow C++ builds and demonstrates how unity build can be an efficient solution. The statement is then assured by measurements. I list different kinds of pitfalls of the technique and offer workarounds. Then I present a method that detects accidental semantic changes caused by applying unity build. In this chapter related work is not separated, instead it is part of the discussion in 2.3 and in 2.6.

2.1 Motivation

The C++ language inherently suffers from slow builds [33]. There is an order of magnitude difference compared to other general purpose object oriented programming languages [33], like C# or Java. Some current Java IDEs can be configured to build the project on each save [48]. This would not be feasible with major C++ code bases.

Build time influences many aspects of software development. Developers often work in a develop-test-develop-test cycle: they modify the code and check if it passes the tests [49]. One key requirement of the continuous integration development strategy is fast auto tests which require fast builds [50]. If the change is not about bugfixing, but creating a new component from scratch, developers check compilability from time to time. Not only the pure time spent on waiting for builds is what counts. Having to wait demotivates and makes the developer start another task which incurs context switch costs.

It is worth differentiating between incremental and full builds. Regular reference builds are usually full builds, that is building the system from scratch, while incremental builds are incorporating the changes since the last successful build. From a developer’s point of view incremental build is important for keeping the
develop-test cycle short. On the other hand as the number of developers on the same code base grow, bigger and bigger or even full build is needed when the working copy is synchronised with the revision control repository. According to common practice this is done at least once each day. This means that full build time is also a concern for developers.

2.2 Root cause of slow C++ builds

Figure 2.1 shows the flowchart of C++ compilation. Compilation can be further split to parser, optimiser and code generator steps. Contrary to most other general purpose languages C++ does not (yet, see later in 2.3) have an explicit module concept. We can call a compilation unit a module, but that is only a physical container. To use a component, the user code has to somehow see its declarations either by repeating them explicitly or \#includeing the header file. The former violates the DRY principle, so inclusion is the preferred way. Inclusion in C++, however, refers to a physical file, not a module. The content of the referenced file is simply made part of the host file and sent through the whole chain of the compilation process just as if they were actually part of the host file. If a component is used in \( n \) compilation units, the header file containing its declarations has to be included in all these \( n \) compilation units and will be preprocessed and parsed \( n \) times. Optimisation and code generator phases of header file processing are typically negligible, because declarations do not result in much generated code. In case of generative programming (especially TMP), heavy use of inlining or whole program optimisation [51, 52], however, we have to count with optimisation and code generation costs for headers as well. In [53] Veldhuizen outlines five different compilation models for C++ templates with different tradeoffs of compile time, code size and code speed. These models range from purely dynamic typing to profile-directed template instantiation.

A typical compilation unit can be devided into an \#include section and an effective code section. The effective code section contains the actual code we want to compile, while the \#include section is only for bringing in the declarations of the
dependencies. I chose an open source C++ library for a case study: OpenSceneGraph [54], a general purpose 3D visualisation engine. It has nicely modularised source code that follows object oriented and C++ guidelines. Figure 2.2 shows the ratio of the \#include and actual code sections in OpenSceneGraph before and after preprocessing. This means that though the \#include section seems to be

<table>
<thead>
<tr>
<th>#include ...</th>
<th>+2% (\rightarrow) preprocessor</th>
<th>+29000%</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual code</td>
<td>100% (\approx)</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 2.2: Size of the \#include section compared to the actual code section before and after preprocessing in OpenSceneGraph.

a little part of the source code compared to the actual code, its actual tokens to compile after preprocessing is roughly 300 times more than those of the actual code section. In other words, only 0.34% of tokens compiled belong to actual code. A similar comparison was performed for further two open source libraries, see later in 2.4. This ratio is far from optimal, and clearly shows there is much space for improvement. While none of the other phases of compilation shows this level of inefficiency, compilation phase contributes most to the full build time. Based on my measurements (see 2.4) we can claim that the \#include mechanism is a key factor in long build times.

2.3 Methods for reducing build times

In this section we enumerate the existing approaches towards shortening build times.

Precompiled headers [55] are like saved compiler states, activated in all or in the selected units at the beginning of the file. We can identify some key header files to be (pre)compiled once, and then be reused without actual compilation in many units. There is a trade-off what to put into the precompiled header. The more headers are included in it, the more we gain in full builds, but the more probability we have in incremental builds that we touch the precompiled header and recompile all its user units. So the expected value of incremental build time starts to increase as we start putting too many inclusions into the precompiled header [56]. In [57] a Makefile-transparent precompiled header automation method is presented, which was successfully implemented in the aCC compiler.
Loose coupling [58] that is fewer dependencies among modules can reduce the amount of inclusions and thus reduce build time. To achieve loose coupling the developer may have to restructure the code by cutting unwanted dependencies. Different visualiser tools can help in revealing the actual dependencies.

IncludeManager [59] is an add-in for Visual C++ that shows a graphical view of \#include file dependencies. The live view of the \#include graph gets updated instantly if the user changes the \#include relations in the code. Build impact graph displays a breakdown of how many preprocessor tokens the compiler should process during compilation. A tabulated view is available for detailed analysis, sortable to display most \#included files, files with most build impact etc.

Build Analyzer [34] is a standalone application that performs build cost analysis and allows browsing its result and the corresponding dependencies. When the developer selects a “fat” header, a bottleneck of the build process, its refactoring view offers optimal splits. An optimal split of the symbols in the selected header file is determined forming two symbol groups to be placed in two separate headers instead of the original “fat” one. The same split procedure is applied recursively on the smaller symbol groups. The refactoring view displays this described split tree visualising possible benefits and costs of the hypothetical refactorings.

In [60, 61] a (semi-)automatic “componentization process” is presented that reclusters the dependency graph so that it has optimal build performance. The algorithm works with a mapping between actual dependencies and high-level architectural clusters. Architectural clusters are defined by the developers. A clusters (describing how the header files are structured) is optimal, if the difference between actual dependencies and architectural clusters is minimal.

In [62] authors argue that Kruchten’s “4+1” view model of software architecture is should be extended by a build-time software architecture view. This additional view is to describe the build process, including the build steps and the dependencies. In case of generative systems with multiphase compilation process this architectural view can clearly describe the non-trivial process of compilation, the rules of transforming source files to build artifacts. BTV Toolkit is a tool that can automatically extract information related to the build process from the make system. The analyzer engine constructs the build-time view from these extracted build time facts and allows visualisation, exploration and comparison.

The pimpl idiom is a special form of the handle/body idiom to act as a compilation firewall. Normally protected and private members and methods of a class are listed in the class definition. Even though the public interface may have not changed, each user code of this class is recompiled whenever the smallest detail is changed in the protected or private declarations. The pimpl idiom reduce these unwanted compilation dependencies by moving protected and private members and methods to another class that is only forward declared on the inter-
The interface class becomes a handle that delegates the actual tasks to the implementation class.

**Parallelised/distributed/cloud builds** apply the divide and conquer method. In a parallelised build the compilation of different compilation units are performed on different threads. Running multiple compilations simultaneously can significantly increase the memory need of the build process, therefore the number of compiler threads has to be chosen wisely. Distributed builds extends parallelisation from multiple CPUs on the same computers to involving multiple computers on the same (local) network. A cloud based solution [69] can serve build requests arriving in bursts with big computation power shared among different groups of users.

**Unity build** [70] exploits the property of typical header files that they are parsed only once in a compilation unit, even if they are `#included` many times. This is achieved by header guards [68] or with `#pragma once` [71]. Where unity build is applied, compilation units with the same compilation flags are `#included` into a single file, and compiled with one invocation of the compiler instead of processing each unit separately. Here the benefit comes from not including and therefore processing the same headers many times in different units. If there are $n$ units using the same set of $m$ headers, their contents are not processed $n$ times, but only once. As we saw that one significant cause of long builds is the `#include` section. We can benefit much from eliminating the redundant `#include` processing in multiple compilation units. FASTBuild [72] is a build framework that supports unity build. Unity build is sometimes called blob build [72].

**Native module support in C++** The C++ standardisation committee has a working group dedicated to native support for modules in C++. After previous suggestions [73] and experimental prototypes in clang [74, 75] the latest version of the concept is summarised in [76]. The fundamental goals are componentisation, isolation from macros, scalable build and support for modern semantics-aware developer tools. Semantic modules introduce the `import` keyword, which replaces raw textual interface import (`#includes`) with a more direct semantic import mechanism. Importable interfaces are exported with the already existing `export` keyword.

## 2.4 Measurements

The main focus of this chapter is the unity build technique. It can be combined with the other approaches, further improving build times. The modified flowchart
of compilation is shown in Figure 2.3. We benefit from not processing the same

![Flowchart of C++ compilation with unity build.](image)

headers multiple times as presented in Figure 2.4. The idea sounds good in theory,

![Multiple processing of the same header files are eliminated by unity build.](image)

but let us see actual numbers. This time two additional open source libraries have been involved in the test. **wxWidgets** is a platform-independent windowing toolkit, while **Xerces** is an xml parser library. Table 2.1 shows preprocessing overhead without and with unity build and Table 2.2 lists the corresponding full build times.

The transformation successfully reduced the preprocessor overhead in each case. While this resulted in approximately 1/10 build times for OpenSceneGraph and Xerces, we cannot say the same for wxWidgets. Either wxWidgets had already an optimal build speed, or its transformed version compiles much slower. Table 2.3
2.4. MEASUREMENTS

<table>
<thead>
<tr>
<th>Library</th>
<th>original LOC</th>
<th>preproc. LOC</th>
<th>preproc. impact</th>
<th>impact ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSceneGraph</td>
<td>91205</td>
<td>16366213</td>
<td>17944%</td>
<td>4.90%</td>
</tr>
<tr>
<td>OpenSceneGraph unity</td>
<td>802053</td>
<td>879%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wxWidgets</td>
<td>357642</td>
<td>40550041</td>
<td>11338%</td>
<td>3.85%</td>
</tr>
<tr>
<td>wxWidgets unity</td>
<td>1562461</td>
<td>437%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xerces</td>
<td>122396</td>
<td>2398884</td>
<td>1960%</td>
<td>9.87%</td>
</tr>
<tr>
<td>Xerces unity</td>
<td></td>
<td>236810</td>
<td>193%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Preprocessing overhead without and with unity build. Note that the ratio for OpenSceneGraph differs from the ratio in Figure 2.2. The difference comes from the type of LOC measure used. Figure 2.2 uses raw physical LOC, while in Table 2.1 LOC figures do not count blank lines.

<table>
<thead>
<tr>
<th>Library</th>
<th>original build time</th>
<th>unity build time</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSceneGraph</td>
<td>1954</td>
<td>196</td>
<td>10.0%</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>253</td>
<td>115</td>
<td>45.5%</td>
</tr>
<tr>
<td>Xerces</td>
<td>204</td>
<td>22</td>
<td>10.8%</td>
</tr>
</tbody>
</table>

Table 2.2: Full build times in seconds without and with unity build.

show the LOC/s ratios, that is the normalised compilation speeds. This number reflects how many lines are processed in one second on average. Note that this pace is not even during the compilation. Effective use of precompiled headers produce short but high peaks, which raise the average even if the compilation speed of the actual code section is about the same. The figures show that wxWidgets already

<table>
<thead>
<tr>
<th>Library</th>
<th>preproc. LOC</th>
<th>build time</th>
<th>LOC/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSceneGraph</td>
<td>16366213</td>
<td>1954</td>
<td>8376</td>
</tr>
<tr>
<td>OpenSceneGraph unity</td>
<td>802053</td>
<td>196</td>
<td>4092</td>
</tr>
<tr>
<td>wxWidgets</td>
<td>40550041</td>
<td>253</td>
<td>160277</td>
</tr>
<tr>
<td>wxWidgets unity</td>
<td>1562461</td>
<td>115</td>
<td>13587</td>
</tr>
<tr>
<td>Xerces</td>
<td>2398884</td>
<td>204</td>
<td>11759</td>
</tr>
<tr>
<td>Xerces unity</td>
<td>236810</td>
<td>22</td>
<td>10764</td>
</tr>
</tbody>
</table>

Table 2.3: Normalised compilation speeds.

had a very good normalised compilation speed, so this is the explanation of the less significant speedup when the unity build technique is used.
CHAPTER 2. ANALYSIS OF UNITY BUILD

The measurements were carried out on the same single core computer several times, then averaged. Apart from the higher memory load caused, this approach is expected to scale similarly to normal builds in a multicore environment. Measuring multicore build performance could be a valuable extension of this analysis in the future.

Note that though the full build times are very nice, the lower limit of incremental build time is much higher. A developer who works mainly on a set of implementation files and rarely touches headers can experience unity build as a stepback in terms of average incremental build times. On the other hand if header file modifications are a bit more frequent, the faster build when multiple compilation units are processed can overweight this disadvantage. A good approach is implementing unity build as a non-compulsory option in the build system.

2.5 Drawbacks of unity build

Though the build time reduction achieved by unity build is attractive, it is not for free. There are several drawbacks of the technique. This section explains these drawbacks and gives recommendations on how to workaround them.

2.5.1 Increased compiler load

Though the merit of unity builds is reducing the amount of tokens the compiler should process, still it causes increased compiler load. The mere number of tokens is less, but the ratio of declarations versus actual code shifts significantly to actual code. That means more effective code to generate in a single compilation unit. The effects of this are increased memory load and additional costs due to using the same internal algorithms of the compiler in greater magnitudes. If there are asymptotically slower than linear algorithms inside the compiler, then the compilation of the united actual code section can be slower than compiling them separately.

It is very important to keep memory load under control. If the OS has to use disk for memory due to page faults, compiler efficiency can drop down as low as 0.1%. The more parallel threads are used for compilation, the lower peak memory usage we should accept to prevent swapping.

So on one hand we can exclude the multiple parsing cost of the headers, but we include processing cost overhead due to having more complex compilation units.

Increased per unit complexity renders some features of the compilation environment practically unusable:

- incremental linking [77] is optimised for compilation units having a limited amount of generated code
2.5. DRAWBACKS OF UNITY BUILD

- edit and continue [78, 79] gets unusably slow with compilation units having big effective code

2.5.2 There is no such as local

It would be nice if we could treat unity build as a technique to speedup our builds, but otherwise we could develop our software just the same way as without using it. Unfortunately unity build cannot be that transparent. It destroys the original meaning of compilation unit, more precisely the united files will be the compilation units for the compiler instead of the original source files. I will use the terms source file and united file to distinguish the two levels of units, the original and the with-unity compilation units respectively. Two main features are lost: independence and locality. Independence means that the processing of a source file does not depend on the contents of other source files. Locality means that we can have declarations (in a broader sense covering macro definitions too) in our source files that are guaranteed to apply only for that specific source file, but not for others. These two notions are the two opposing viewpoints of multiple source files being compiled as one. Figure 2.5 shows the relation.

Figure 2.5: Loosing independence and locality in unity builds.
CHAPTER 2. ANALYSIS OF UNITY BUILD

Typical problems of unification

The following list contains some typical problems caused by uniting. These cover the cases I encountered when applying the technique to the mentioned open source projects and also some industrial projects, but this does not mean that the list is complete.

Prefixed inclusion  Some `#include` files allow preconfiguration with macros. For example `windows.h` has a lightweight version that can be enabled by defining `WIN32_LEAN_AND_MEAN` before `#include <windows.h>`, see Figure 2.5. It is important to prevent local macros from falling through to the next source file to avoid misconfigured `#include`s.

using declarations  Global `using` declarations cannot be undone. That means once we brought some symbols into scope, they remain there in the forthcoming source files as well. Using declarations embedded in some nested scope are not so harmful.

Conflicting headers  There can be headers that do not mix well. In one of them `min` can be implemented as a macro, while in the other it is a (template) function. Many times this can be worked around by resolving the specific conflict, e.g. by `#undefining` the `min` macro in our example. If this is not an option, we may have to accept the conflict and put the host `.cpp` files into separate united files.

Conflicting file local symbols  File local symbols often have short names, and therefore clash easily. Fortunately renaming them is always a relatively easy option.

ODR violation  The one definition rule (ODR) [81] does not allow multiple definitions of the same non-inline function not only in the same translation unit, but also in the whole program. For inline functions multiple definitions are allowed, but only in different compilation units. By unifying these different units we will violate ODR. I encountered this problem when united the `float` and `double` instantiations of the `Matrix` template. It was not a real `template`, but was implemented by `#includeing` the same implementation with contextual macros in it defining the actual type to use (see Listing A.1 for a simplified version). This technique of simulating generics in C is described in more detail in 3.1.1. The implementation file contained the definition of an inline function that did not depend on the type, its signature was the same in both units. This led to a compilation error of violating ODR. Adding a header guard-like protection to prevent multiple
inclusion of the same function definition solved, and in general solves this kind of issue reliably.

**Changed behaviour** We have to feel lucky if uniting produces a compilation error. It is always much better because we have been notified about the problem. Sometimes, however, we are not that lucky, the resulting code silently behaves differently with unity build. This can be the side effect of a dangling macro, or function overload. Once the difference got identified, it can be handled just like the other, more visible conflicts. The challenge here is the detection. A decision can be to intentionally skip prevention hoping that the costs of fixing the occurring problems is cheaper. A safer approach can be the comparison of the code generated by normal and unity builds. Determining whether the generated codes are semantically equivalent is very difficult, because at that point, after optimisation, the semantics are hardly recognizable. Because of low level optimisations functionally equivalent programs can have big differences in their generated code. Another approach is comparing the internal structures of the compiler in normal and unity builds. This can be an option if the corresponding internal structures that describe semantics are accessible during compilation. Section 2.6 discusses this kind of equivalence check in more detail.

**Exploitation of implementation details** In unity builds an originally file local symbol can be used in another implementation file without being noticed or causing functional errors. That symbol, however, was probably not made file local by accident, it was rather a design decision. Further changes may build on that initial design, and does not take the other usage into account. It is important to eliminate these hidden dependencies, unity build should not alter the original design.

**Workarounds for problems caused by unification**

Here I recommend several techniques that can reduce the number of the problems listed above.

**Make macros local** #undef them at the end of the source file. While this is an easy rule to follow in case of locally defined macros, it is not trivial to list all the definition brought in by #includeing some headers.

**Do not use global using declarations** Use explicit scoping instead.
Explicitly handle conflicts  Add header guard-like construct to eliminate double definition of inline functions, \texttt{#undef} macros brought in by headers, put \texttt{#include} of conflicting headers to separate united files etc.

Keep separate build an option  To be able to compare the original and united versions of the build and to eliminate hidden dependencies, it is advisable to implement unity build in an optional manner. A simple way of doing that is introducing a unity build toggle define, say \texttt{UNITY_BUILD}, and framing all the source and united files with guards. This way both the source and united files can remain part of the build process, but only one kind of them will be actually compiled. As the source files should be part of the compilation even in case of unity builds, the flag should be disabled for the time of their \texttt{#include} to deactivate the guard that prevents their standalone compilation. See Figure 2.6 for an example. To perform my measurements on the listed open source libraries,

\begin{verbatim}
x_all.cpp
#include "x1.cpp"
#include "x2.cpp"

x1.cpp
#ifndef UNITY_BUILD
...
#endif

x2.cpp
#ifndef UNITY_BUILD
...
#endif
\end{verbatim}

Figure 2.6: Unity build as an option.

I had to make the code base unity build capable. I did not added the toggle, but tested the sample applications to somewhat make sure that the unity build version has the same functionality as the original version. Table 2.4 shows the amount of code modifications I had to perform. That means I had to adapt 2-10\% of the

\begin{table}
\centering
\begin{tabular}{|l|c|c|c|c|}
\hline
Library & Files & Modified files & Characters added & Average added characters per modified file \\
\hline
OpenSceneGraph & 1661 & 31 & 2418 & 78 \\
wxWidgets & 825 & 82 & 12100 & 148 \\
Xerces & 811 & 50 & 5082 & 102 \\
\hline
\end{tabular}
\caption{Amount of changes needed to make the libraries unity build capable.}
\end{table}
2.6. STRICT EQUIVALENCE CHECKS

source files. Most of the changes were only renames.

2.5.3 Errors caused by automated unification

Some projects already #include .cpp files if, for example, generics are simulated via macros as in the example above for ODR violation (see also 3.1.1). If scripts are used to unite source files based on file system listing, these already #included files should be skipped and should not be #included directly from the united file. A good practice for this exclusion can be to distinguish these files with a regex-recognisable naming convention. Some use simply a different extension (.inc), but that may mislead the text editor and thus disable syntax highlight.

2.5.4 IDE wants to compile all implementation files

In integrated development environments (IDE) the implementation files usually have to be added to the project to enable code completion and other useful features on them. If, however, they are added to the project, the build process will build all of them even if they are already #included in the united files. We should ensure that the source files are excluded from the build process not to build the whole system twice, once the source files separately, and also the united files, and having all the symbols doubly defined producing a long list of linker errors. Either we can use an IDE specific way to exclude the source files from build, or branching on toggle defines as described above related to keeping separate build an option.

2.6 Strict equivalence checks

In 2.5.2 we saw that the transformation could change the behaviour of our program. While better build times are very attractive, even a minimal risk of unnoticed changes can be unacceptable. In this section a method is presented to compare normal and unity build compilations to catch possible semantic differences. The recommended setup is having the unity build toggle implemented and executing the described equivalence checks regularly on a build server as part of the QA process.

There are three individual checks to be executed one after the other: unit structure equivalence ($EQ_u$), token sequence equivalence ($EQ_t$), and ABT equivalence ($EQ_a$):

$$EQ := EQ_u \land EQ_t \land EQ_a$$

$$EQ(\text{united, original units}) := \bigwedge_{E \in \{EQ_u, EQ_t, EQ_a\}} E(\text{united, original units})$$
Consider the example in Figure 2.7. Here the \texttt{#include} hierarchy of the original compilation units has multiple levels. In such a situation we can perform equivalence checks recursively:

\begin{align*}
EQ(a\_all, \{a1, a2, a3\}) \\
\land EQ(c\_all, \{c1, c2, c3\}) \\
\land EQ(all, \{a\_all, b, c\_all\})
\end{align*}

or all at once:

\[ EQ(all, \{a1, a2, a3, b, c1, c2, c3\}) \]

The two approaches are equivalent, because both \textit{EQ} and \texttt{#include} are transitive operations.

\subsection{EQ\textsubscript{u}}

Execute the preprocessor on all units including the original ones (\textit{unit\textsubscript{1}}, \ldots, \textit{unit\textsubscript{n}}) and also the united one (\textit{unit\textsubscript{\textomega}}). Let \( f_i \) denote the file path of \textit{unit\textsubscript{i}}, while \( F \) the union of them:

\begin{align*}
f_i &:= \text{file path of } \textit{unit\textsubscript{i}} \\
F &:= \{ f_i | i \in [1..n] \}
\end{align*}
2.6. STRICT EQUIVALENCE CHECKS

The result of preprocessing is a finite sequence of (file path, line number, token) triplets:

\[
tokens_i \in (\mathcal{F} \times \mathbb{N}_+ \times \mathcal{T})^* \\
t_i := \max D_{tokens_i}
\]

where \( \mathcal{F} \) is the set of possible file paths and \( \mathcal{T} \) is the set of C++ tokens.

\[
EQ_u := \forall i \in [1 \ldots n], j, k \in [1 \ldots \tau], tokens_{\cup, f}(j) = tokens_{\cup, f}(k) = f_i : \\
norm{\{tokens_{\cup, f}(l) | l \in [j \ldots k]\} \cap \mathcal{F}} = 1
\]

or alternatively

\[
EQ_u := \forall i \in [1 \ldots n], j, k \in [1 \ldots \tau], tokens_{\cup, f}(j) = tokens_{\cup, f}(k) = f_i :\\
\forall l \in [j \ldots k] : (tokens_{\cup, f}(l) \in \mathcal{F} \rightarrow tokens_{\cup, f}(l) = f_i)
\]

The \( EQ_u \) criterion does not allow tokens of different \( unit_i \)'s to mix. In the original compilation of \( unit_i \) it is impossible to encounter a token of any other \( unit_j \). Consequently that section of \( unit_{\cup} \) which is responsible for compiling \( unit_i \) should not contain any token of any other \( unit_j \). Note that this criterion does not say anything about tokens originating from other source files such as headers.

2.6.2 \( EQ_t \)

Definition. Filtered sequence

\[
f \in A^* \\
P : A \rightarrow L \\
f \cdot P := f \circ i_f^P
\]

where

\[
i_f^P(0) := 0 \\
i_f^P(j) := \min\{k > i_f^P(j - 1) | P(f(k))\} \quad (j > 0)
\]

That is the filtered sequence of \( f \) by \( P \) contains only those elements of \( f \) that satisfy \( P \), in their original order.

\[
EQ_t := \forall i \in [1 \ldots n] : tokens_i \cdot (f = f_i) = tokens_{\cup} \cdot (f = f_i)
\]

By using new symbols for the filtered sequences:

\[
ftokens_i := tokens_i \cdot (f = f_i) \\
ftokens_i' := tokens_{\cup} \cdot (f = f_i)
\]
the criterion can be rewritten as

\[ EQ_i := \forall i \in [1..n] : f\text{tokens}_i = f\text{tokens}'_i. \]

\( EQ_i \) evaluates to true if and only if all the tokens that originate from a given \( unit_i \) appear in \( unit_\cup \), in the same order, and there is not any additional or missing \( unit_i \) token in \( unit_\cup \).

**Theorem.**

\[ EQ_u \land EQ_t \rightarrow \exists \sigma \in S_n : tokens_\cup \cup (f \in F) = \| \ f\text{tokens}_{\sigma(i)} \]

where \( S_n \) is the set of permutations of \([1..n]\) and \( \| \) denotes finite sequence concatenation. That is if we exclude every token not directly from \( unit_i \) (tokens of \#included headers etc.) then the remaining token sequence of the united file should be the concatenation of the remaining token sequences of the individual units in a certain order. The theorem can be proved by going through \( tokens_\cup \cup (f \in F) \) and continuously constructing \( \sigma \).

**Proof.** \( EQ_u \) ensures that \( \exists \sigma \) such that \( tokens_\cup \cup (f \in F) \) is the concatenation of arrays of repeated \( f_{\sigma(i)} \). \( EQ_t \) ensures that the corresponding token arrays in \( tokens_\cup \cup (f \in F) \) actually contain \( f\text{tokens}_{\sigma(i)} \). \( \square \)

\( EQ_i \) reliably detects if a dangling macro definition from a previous unit results in an unwanted macro expansion in a following unit.

### 2.6.3 \( EQ_o \)

To understand \( EQ_o \) first we have to understand what ABT is. The process of C++ compilation can be divided into several logical phases. First the lexer interprets the character sequence and forms tokens. These tokens are fed to the preprocessor which performs macro expansions, file \#includes etc. Then the parser identifies higher level syntactical elements in the raw sequence of tokens by applying the rules of the grammar. The result of this syntactical analysis process is the abstract syntax tree (AST) [82], that describes the syntactical structure of the compiled code. AST is still more about syntax than semantics. Semantical analysis transforms an AST to an abstract binding tree (ABT), where the exact relations of symbols are already resolved. In the ABT of a simple assignment statement, for example, we find the name of the target variable in an identifier token. In the corresponding ABT we find a link to the declaration of the target variable. Intermediate code is generated from the ABT, which is then optimised. Then native code is generated from intermediate code, involving further platform specific optimisation steps.
The described scheme was a theoretical overview of the internals of a C++ compiler. In practice these phases are rarely that distinct. Lexical analysis and preprocessing are usually strongly integrated, but that is also true for syntactical and semantical analysis. These unclear boundaries partly come from the properties of the language, e.g. C++ is context-sensitive [83]. In other cases there is a historical performance consideration in the background [84, 85]. Some compilers, clang for example, provide access to the ABT. There it is called AST [86], presumably because AST is a more common notion, and the joint execution of syntactical and semantical analysis does not allow easy distinction anyway. Note that if we treat the backreferencing links (the link to the declaration of the variable in our example) part of the structure, then the ABT becomes a (directed acyclic multi-) graph, the abstract binding graph (ABG) [87]. As ABT is often called AST, ABG is often called ASG (abstract semantic graph) [88].

The $EQ_a$ criterion checks ABT equivalence of $ftokens_i$ and $ftokens_i'$. An ABT has nodes of different types ($=: T_v$). Each node has a set of attributes differentiated by attribute keys. Between the nodes there are edges of different types ($=: T_e$). Some of the attributes contain such data that is not in direct connection with the semantics of the compiled program. In case of file local variables (in unnamed namespaces or marked static), for example, the compiler generates a unique name for the symbol [89]. Though these generated symbol names may differ in independent builds of the same code, this difference does not have any effect on the behaviour of the generated code. Consequently if $EQ_a$ compared the corresponding ABTs directly, we would get a false negative result for the equivalence. To support such cases I created the criterion with customisation in mind. Instead of comparing all existing information, the criterion only compares narrowed subsets of node types, relations and attributes. In the following formal description ABT nodes are referred to as vertices.

\[
T_v := \text{ABT vertex types} \\
T_e := \text{ABT edge types} \\
ID_v := \text{unique (within an ABT) vertex identifiers} \\
K := \text{ABT vertex attribute keys} \\
W := \text{ABT vertex attribute values} \\
\mathcal{V} := T_v^{\text{type}} \times ID_v^{\text{id}} \times (K \rightarrow W) \quad \text{ABT vertices} \\
\mathcal{E} := T_e^{\text{type}} \times \mathcal{V} \times \mathcal{V} \quad \text{ABT edges}
\]

Let $A_i$ denote the ABTs of the different units:

\[
A_i := (V_i, E_i) := \text{ABT}(tokens_i) \quad (i \in \{1, \ldots, n, \cup\})
\]
where \( V_i \subseteq \mathcal{V} \) are typed vertices and \( E_i \subseteq \mathcal{E} \) are typed edges.

\[
\alpha_i : [1 \ldots t_i] \to V_i
\]

\[
\alpha_i(j) := \text{the narrowest ABT vertex in } A_i \text{ containing } \text{tokens}_i(j)
\]

By the nature of the ABT \( \alpha_i \) is a function. For each token there is only one ABT vertex where the given token directly belongs. Note that the opposite is not true, an ABT vertex can directly represent multiple tokens.

\[
T_v^- \subseteq T_v
\]
\[
T_e^- \subseteq T_e
\]
\[
K^- \subseteq K
\]

are the narrowed subsets to consider in the equivalence checks.

\[
\mathcal{V}^- := T_v^- \times I D_v \times (K^- \to W)
\]
\[
\mathcal{E}^- := T_e^- \times \mathcal{V}^- \times \mathcal{V}^-
\]

\[
V_i^- := V_i \cap \mathcal{V}^-
\]

\[
E_i^- := E_i \cap \mathcal{E}^-
\]

\[
A_i^- := (V_i^-, E_i^-)
\]

\( A_i^- \) is the equivalence subgraph of \( A_i \) containing only those information that should be compared by the equivalence checks.

**Definition.** Closed neighbourhood

\[
N_G[X] := X \cup \{v \in V(G) | \exists u \in X : uv \in E(G)\}
\]

**Definition.** Reachable vertices

\[
\]

with such \( k \) that

\[
N_G^{k+1} = N_G[N_G^k[X]] = N_G^k[X]
\]

**Definition.** Induced ABT

\[
I_i(f) := (U, E_i^- \cap U \times U) \quad (i \in \{1, \ldots, n, \cup\})
\]

where

\[
U = \bigcup_{j=1}^{i} R_{A_i^-} [\alpha_i(j) \cap V_i^-]
\]
The induced ABT of $f$ contains those ABT vertices in the equivalence subgraph of $A_i$ that are reachable from tokens of $f$.

$$EQ_a := \forall i \in [1 \ldots n] : \exists m : ID_u \rightarrow ID_v : I_i(f_i)^{id\cdot m(id)} = I_\cup(f_i)$$

The criterion compares the induced ABT of $f_i$ tokens in $A_i^\cup$ with the induced ABT of $f_i$ tokens in $A_i^\wedge$. Here ABTs come from different compilation sessions, therefore comparing the identifiers directly between these independent ABTs does not make sense. The criterion checks if the node types, the typed connections among them and the attributes are exactly the same. The check is performed only on the narrowed subsets of these properties.

It is important to see that the induced ABTs are only a small subset of the whole ABT that is built during parsing. Many declarations and definitions coming from the \#included header files are processed in a typical build, but only a small portion of them is actually referenced. $EQ_a$ deals only with these actually referenced items and compares them to the deepest details thoroughly. This check, however, does not say anything about the unreferenced parts of the ABT. Passing the $EQ_a$ test does not mean that there are not any symbols visible in $f_i$ that should normally be hidden. The test is always about the actual versions of $f_i$ as they are, and the test result should not be used to predict unification correctness of any other, modified set of $f_i$, whatever little modification it is.

### 2.7 Correctness and applicability

The $EQ$ check seems very accurate and thorough, which leads to questioning if it is a sufficient or a necessary condition for a correct unification. Let us discuss the two parts separately:

$\Rightarrow$: Every unification that passes $EQ$ is a correct unification.

$\Leftarrow$: Every correct unification passes $EQ$.

\_\_COUNTER\_\_ is a common predefined macro that expands to increasing integer values (starting from 0) whenever it is referenced. It is an ideal tool for creating unique identifiers in macro generated code. While in normal builds this counter starts from 0 in every unit, in the unified build every occurrence has a unique value not only on unit level, but for the whole scope of unification. $EQ_a$ will signal the difference, which is good, because the actually compiled source code is really different. On the other hand when \_\_COUNTER\_\_ is used solely to differentiate macro generated symbols that are by design not referenced by user code, which is often the case, we may want to ignore this little difference and declare the tokenwise different compilations semantically equivalent. Kleene equalence [87]
for example means that for the same input the two programs give the same output (observational equivalence), which has nothing to do with tokens. Depending on how we define the correctness of a unification the above \texttt{\_COUNTER\_} case can or cannot fail $\Leftrightarrow$.

In 2.6.3 I emphasised that $EQA$ does not cover anything that is not directly referenced from within $f_i$. The whole idea behind unity build is not to compile the same headers many times, consequently there surely will be differences in how the two build approaches process them. This is the very reason why the \texttt{\#included} headers only indirectly appear in the checks. The main concept of unity build assumes that we can freely optimise the build process by reusing compilations of otherwise multiply compiled header files. If the compilation of these header files results in anything more than just intermediate information for the rest of the compilation process, some generated code for example, and does so even if the corresponding code is not referenced, then the unity build is not applicable. $EQ$ is not capable of detecting this situation, which means that $\Rightarrow$ is not universally true.

Definition. direct $f_i$ code
$f_i$ without its \texttt{\#includes}.

Hypothesis. $EQ$ completeness
Let us assume that we treat differences in the compiled tokens corresponding to direct $f_i$ code an incorrect unification even if the same code is generated semantically.
Furthermore let us assume that compilation of header files produces only intermediate information for the compiler, and code is generated only if the corresponding symbols in the header files are referenced from the client code.
Provided that these two assumptions are met a unification passes the $EQ$ test if and only if the unification is correct.

I verified $EQ$ on the listed typical problems (see 2.5.2) as well as on artificial cases and the hypothesis passed all of them. I strongly believe that the $EQ$ completeness hypothesis is true, but could not yet prove it formally.

I intentionally kept the discussion of the $EQ$ test formal to be as exact as possible, but free from implementation specific details. The exact format of the ABT, if ABT dump is supported at all, differs from compiler to compiler. I recommend implementing the $EQa$, $EQt$ and $EQa$ checkers in this order building on each other. $EQa$ ensures that token sequences do not overlap in $f_i$, which can be exploited in the implementation of $EQt$. In $EQa$ finding the appropriate identifier mapping ($m$) between the induced ABTs may seem to require a possibly long search procedure at first glance. In fact if we know that $EQt$ is true, then we can
easily process the same tokens in the two compared units (one normal and the corresponding part in the united) one by one, continuously building up the mapping while simultaneously traversing the two graphs. A depth first search that marks the visited nodes should suffice. Clang directly supports the recursive traversal of its AST with the `RecursiveASTVisitor` class \cite{86} for example. If a difference is found in any of the important types, links or attributes, the traversal can end. Otherwise the structure should be exactly the same, therefore the mapping can be continuously created for newly encountered references, or checked for the existing ones.

2.8 Evaluation

According to the measurements unity build can be an effective tool for reducing compilation time. This is definitely true for full build times, and probable for average incremental build times as well if header modifications are not too rare. The technique is showed to significantly reduce build time even when precompiled headers were already effectively used for the same purpose. Despite its advantage the technique is not very well known. The idea of \#include\ing together otherwise unrelated implementation files is a bit bizarre, and in fact has many disadvantages. Code modifications are needed to make the code base compatible with unity build. We have to give up using some language features, or have to use them cautiously to prevent compilation errors or changed behaviour. There are techniques to preserve standalone builds as an option, and automatic detection of changed behaviour is possible by comparing the results of the two builds. This comparison can be implemented easily, and can catch every defect of unification that I can imagine, provided that the prerequisite of the unification is met.

2.9 Contribution

**Thesis 1** (Analysis of unity build). *I analysed the unity build technique and measured its effectiveness to reduce build times focusing primarily on full build times. Based on case studies with three open source libraries I determined the pros and cons of applying unity build on existing projects and derived recommended approaches for its implementation. I defined a set of criteria that allow automatic detection of unwanted silent semantic changes caused by the unification. I also derived an algorithm for the implementation of this equivalence check.*
## Analysis of Unity Build

<table>
<thead>
<tr>
<th>thesis name</th>
<th>relevant publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of unity build</td>
<td>• •</td>
</tr>
<tr>
<td>Template metaprogram instrumentation</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
</tr>
<tr>
<td>Methods for template metaprogram debugging</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
</tr>
<tr>
<td>Methods for template metaprogram profiling</td>
<td>◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦ ◦</td>
</tr>
<tr>
<td>Type-preserving heap profiler</td>
<td>◦</td>
</tr>
</tbody>
</table>
Chapter 3
Generative metaprogramming

We defined generative programming in the introduction chapter (1) as an approach in which carefully generalised software modules are created and then their concrete variations are combined [17]. One way of implementing generalisation is run time branching of execution according to the actual parameterisation. This, however, incurs run time and executable size costs. To eliminate this overhead and let the optimiser exploit all details of the actual usage, the specialisation is often made during compilation. In this approach, the rules of creating the final source code are defined by a program. Programs that manipulate other programs as data are called metaprograms [90]. We speak about generative metaprogramming if generative programming is implemented by metaprograms.

The standards of the C language [91] family prescribe the existence of a program transformation step in the compilation process: the preprocessor. C++ [21] adds another transformation: templates are instantiated during compilation of the preprocessed source, see Figure 3.1. Note that in modern compilation environments compilation and template processing are not separate phases as the figure would suggest, the compiler instead invokes the template processor on demand when a new specialisation is required.

Figure 3.1: Standard code transformation steps in C++. 
The following sections discuss the basics of preprocessing and template instantiation, the two standard areas of generative metaprogramming in C++ as a foundation for the theses in the next chapters.

## 3.1 Preprocessor metaprogramming

The C preprocessor originally served two basic needs:

1. import interface (declarations) of other modules placed in header files as a replacement for a real module mechanism
2. expand macros, that is perform textual substitution

The limited set of requirements resulted in a limited set of language constructs, that perfectly did their job. Table 3.1 summarises what operations can be performed in preprocessing time.

<table>
<thead>
<tr>
<th>operation</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>file inclusion</td>
<td><code>#include path</code></td>
</tr>
<tr>
<td>macro without parameters</td>
<td><code>#define macroname definition</code></td>
</tr>
<tr>
<td>macro with parameters</td>
<td><code>#define macroname(args) definition</code></td>
</tr>
<tr>
<td>macro undefinition</td>
<td><code>#undef macroname</code></td>
</tr>
<tr>
<td>conditional compilation</td>
<td><code>#if condition</code></td>
</tr>
<tr>
<td></td>
<td><code>#ifdef macroname</code></td>
</tr>
<tr>
<td></td>
<td><code>#ifndef macroname</code></td>
</tr>
<tr>
<td></td>
<td><code>#elif condition</code></td>
</tr>
<tr>
<td></td>
<td><code>#else</code></td>
</tr>
<tr>
<td></td>
<td><code>#endif</code></td>
</tr>
<tr>
<td>token concatenation</td>
<td><code>token1_##token2</code></td>
</tr>
<tr>
<td>conversion to string literal</td>
<td><code>#identifier</code></td>
</tr>
<tr>
<td>macro definition test</td>
<td><code>defined macroname</code></td>
</tr>
<tr>
<td>integer arithmetics</td>
<td><code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code></td>
</tr>
<tr>
<td>logical</td>
<td><code>!</code>, <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code></td>
</tr>
<tr>
<td>ternary conditional</td>
<td><code>condition ? truepart : falsepart</code></td>
</tr>
</tbody>
</table>

Table 3.1: C preprocessor language constructs

Besides the direct usage of these preprocessor constructs several patterns become best practice for specific purposes over the years.
3.1. PREPROCESSOR METAPROGRAMMING

3.1.1 Preprocessor patterns

Configuration Configurable systems need a way of specifying the actual configuration. There can be global compile time switches that determine what parts of the code should be enabled for compilation, or which versions to use from a collection of available code fragments. The configuration header is a designated header file for specifying global compilation flags [92]. It is typically a list of macro definitions along with explanatory comments. All source files include this setting file (often through the precompiled header), therefore the settings are available at any source code location. Configuration headers can be hierarchical, where further internal options are set based on external settings. Dependencies and incompatibilities are handled and detected at this internal level.

#include macropath The argument of #include is not required to be a string literal. Sometimes it is useful to make inclusion indirect by hiding the actual file path to #include into a macro. This technique is often used together with hierarchical configuration headers. The external setting chooses an option, then an internal macro is automatically set to a relative file path corresponding to the source code of the selected option. Figure 3.2 shows an example.

```c
// Choose one of the supported encoding options:
//#define STRING_ENCODING_ASCII
#define STRING_ENCODING_UCS32
//#define STRING_ENCODING_UTF8

#include "rules.h"
```

```c
#ifdef STRING_ENCODING_UCS32
#define ENCODING_HEADER "encoding/ucs32.h"
#endif
```

```c
#include "config.h"
#include ENCODING_HEADER
```

Figure 3.2: The #include macropath pattern.
Simulate generics in C with includes If a source file uses macros that are not defined locally, it is actually a parameterised source file, since by changing the given macro definitions externally, we can alter the final contents of the file. With this technique some typical usages of C++ templates can be simulated in C.

Suppose, for example that we would like to implement a generic matrix data structure. Here most of the implementation is the same regardless of the actual scalar type used to store the values of matrix elements. In C++ we could define a template class parameterised by the scalar type. In C we can achieve the same by introducing a macro for the scalar type, and making sure that it is defined properly when the implementation is compiled. If we need multiple instantiations with different scalar types, we have to make these instantiations differ in their names, so we need an additional parameter to customise the actual name. Alternatively we can derive the name by using the scalar type. See Figure A.1 for an example. A variation of this technique is used in OpenSceneGraph [54] to reuse the same matrix implementation for float and double scalar types (see 2.5.2).

The X macro The principle of this pattern [93, 94] is the same as above for simulating generics: a generic code using macros is reused in specialised variants by defining the parameter macros differently. This is actually the instantiation of the generic code. While in the previous pattern the generic code is a whole file, in the X macro pattern the generic part is placed in a (typically parameterless) macro. This pattern is often used for having a single definition of a list, while specialised code fragments are generated for the list items at multiple places of the code base. Refer to Listing A.2 for an example. The X macro was extensively used in the operating system and utilities for the DECsystem-10 as early as 1968 [93].

Direct product Compared to an if (...) ... else if (...) ... chain, switch is a more self-explanatory branching construct. With many options it is also faster [95]. Unfortunately switch can only branch on a single value. Sometimes our cases are defined as a set of values, but the overall information would easily fit into a single integer. In these cases we can consider using a function that maps the set of arguments to one integer and use this transformation in the switch statement either in its condition part and in the cases as well. Since the case values of the switch statement have to be const expressions, it is important for the mapping to be implemented as a macro or a constexpr function [96]. Listing A.2 demonstrates this technique by composing a single color value from its red, green and blue byte components.
3.1. PREPROCESSOR METAPROGRAMMING

```c
#define IDENTITY(...) __VA_ARGS__

#define NOCOPY(name, base_classes) \
    class name : IDENTITY base_classes {
        name(const name&) = delete; \
        name& operator=(const name&) = delete;
    }

NOCOPY(MyClass, (public T<A,B<int,long,void(int,long)>>))

```

Listing 3.1: Workaround for macro arguments that may contain commas.

3.1.2 Pitfalls of using the preprocessor

Though in many cases the preprocessor can be treated as a macro processor on text, it actually operates on tokens. All text it operates on should be a valid sequence of C/C++ tokens. In case of metaprogramming this is not a real limitation.

Tokens that have special meaning for the preprocessor are, however, more problematic. Macro definitions, just like all preprocessor directives, should be in one single line. That means we cannot add newline characters to macro definitions. This makes diagnostics harder, because long preprocessor output cannot be broken to multiple lines, which kills readability. Fortunately long macro definitions can be structured at least in the source code with trailing \-s as demonstrated in Figure A.1.

The , (comma) token has multiple roles, and what makes it more problematic, these roles span across different stages of the compilation. A comma can separate macro arguments, template parameters, template arguments, standalone expressions, declarations, enumeration values, function parameters, function arguments and I am not yet sure the list is complete. Even if we restrict a macro argument to denote a type, the comma problem arises for templates with multiple parameters. As always \[97\] we can workaround by introducing an extra level of indirection: each argument that has a non-zero chance to contain commas has to be passed as a tuple as shown in Listing 3.1 and in Figure 3.3.

```
tuple elements
NOCOPY(MyClass,(public T<A,B<int,long,void(int,long)>>))
NOCOPY arguments
```

Figure 3.3: An extra level of indirection to workaround the comma problem in macros.
CHAPTER 3. GENERATIVE METAPROGRAMMING

3.1.3 Boost.Preprocessor

Among the Boost libraries [22] we can find the Preprocessor package [47] which contains a comprehensive set of tools for preprocessor metaprogramming. Preprocessor level data types are introduced as shown in Table 3.2. We can find iterative control structures, arithmetics, conversions between the different data types. Though the preprocessor can easily evaluate arithmetic expressions, computed values of subexpressions are not substituted textually. This property of preprocessor evaluation makes the arithmetic functionalities of Boost.Preprocessor relevant, where the substitution is performed.

Listing A.3 demonstrates how sequences extend the possibilities of the $X$ macro. With the traditional form of $X$ macro the processing order of the values is defined by the generic code containing them. In a list, for example, we are typically restricted to an elementwise operation that has no information about the other values. With sequences we can freely reference any of the values.

<table>
<thead>
<tr>
<th>name</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>(4, (a, b, c, d))</td>
</tr>
<tr>
<td>list</td>
<td>(a, (b, (c, (d, BOOST_PP_NIL))))</td>
</tr>
<tr>
<td>sequence</td>
<td>(a)(b)(c)(d)</td>
</tr>
<tr>
<td>tuple</td>
<td>(a, b, c, d)</td>
</tr>
</tbody>
</table>

Table 3.2: Preprocessor level data types.

3.1.4 Evaluation

The preprocessor operates on tokens. It ignores all the semantic rules of C++. As a low level source code sculpturing tool, we can use it to programmatically create the final token sequence we want to compile. At this point we can implement such compile time mechanisms that support some kind of generality, or ensure consistency, all without adding any run time cost. At this level we are not restricted to the complex semantic rules of C++ building blocks.

On the other hand, this low level processing method leads to problems when the semantic units of C++ are not recognised. A parameterised template class is, for example, treated as a sequence of macro arguments. Due to the lack of semantic information, preprocessor metaprograms cannot differentiate values and types, classes or functions. The proper usage of the tools to construct a valid C++ token sequence is the programmer’s responsibility. There is no direct error message that signals if our macros are misused.
3.2 Template metaprogramming in C++

When templates were introduced in C++ (early 90’s), their primary purpose was supporting generic programming. Following the fundamental language design principles of C++, orthogonality, avoidance of unnecessary constraints, Template was born as a very powerful tool. Classes, (member) functions can be parameterised with types, compile time scalar values, (pointers to) (member) functions. With self-referring template types recursion can be easily implemented, and template specialisations can be used to terminate these recursions. With these primitives templates form a functional programming language embedded in C++ [44]. They are proved to be Turing complete [45, 46] , though this strength was not exploited for a long time. The first template metaprogram was a code that produced output in the form of diagnostic messages. The artificially triggered diagnostic messages listed the prime numbers during compilation [40, 41]. Based on this technique a whole new area opened in C++ recognising that templates are actually a new level of preprocessing that can be used for non-trivial algorithms similarly to preprocessor metaprograms where macros transform token sequences.

Templates have some key strengths over preprocessor metaprogramming. Template metaprograms operate on types and values, not tokens. The template mechanism is integral part of the language, with strong relation to its type system. Similarly, arithmetic computations are a natural task of the compiler, while in preprocessor metaprogramming it is artificial and therefore limited. Not being forced to write the definition into one single line is also a big convenience over macros. As fundamental parts of the language, templates can be scope-sensitive. It is common to open an internal (sub)namespace, often named detail, to hide the helper templates that are actually implementation details. This cannot be done with macros as they share a single global namespace.

3.2.1 Applications

In this section I discuss applications of C++ template metaprograms. The aim is rather to demonstrate how versatile TMP is than to give a profound introduction. To get a more complete description of the techniques refer to [47].

Active libraries [98] A traditional (i.e. not active) library contains static definition of types, interfaces, and a fixed implementation of the provided interface. Both the interface and the implementation may be adjusted to the actual environment via preprocessor configuration parameters, but after fixing these parameters, the code is actually static. In active libraries the implementation is more dynamic. There can be multiple implementation variants of the same functionality, and an automatic optimisation chooses the optimal variant during compilation, depending
on the actual environment where the library is deployed. If the same functionality is used with different parameters, this automatic optimisation may choose a different variant.

**Compile time computation**  
Let us suppose we are writing a program where the sine of different angles is often needed. Also, performance is a key concern. One solution is using precomputed values in a lookup table. Precomputed values however have a certain resolution, we have values at equidistant positions. For an input between two such points, we can either use the nearest value or an interpolated one of the two neighbours, but neither will be precise. Here we have a trade-off between storage and precision.

It can be a design decision to make this a parameter, especially if we are writing a library for general usage. How can we implement the filling of the values then? One answer can be a simple loop at program startup. This incurs a run time cost, that is probably ok in most cases. Still, if we want to avoid it, we have to compute the values in compile time. One way of this is an extra build phase that generates the table with a script. This solution makes the build process more complicated, which can be painful if we want to support more platforms. With template metaprograms we can implement the computations while staying in the language.

Veldhuizen demonstrated how sine values can be computed in compile time by series expansion in [99].

C++11 introduced generalised constant expressions [96] for the very purpose of performing operations in compile time, therefore this usage of template metaprograms is by now covered by direct language features. Still, there are many legacy code bases that contain such usage or even technical limitations that block their transition to C++11.

**Static assertion**  
Run time assertions are a tool for detecting program errors by placing checks of invariants scattered in the source [100]. Most assertions test function arguments, some properties of the computed result, or a class invariant. Some programming languages (Eiffel [26], for example) directly support these checks facilitating design by contract [101]. In C/C++ `assert` is part of the standard library.

As `assert` in run time programs, `static_assert` [102] serves for detecting unexpected conditions, unsupported combinations of parameters or implementation errors at compile time. The argument of `static_assert` is a compile time expression, which, by the intent or assumption of the programmer, is guaranteed to be true. If the compiler finds that the expression is false, a compilation error is emitted. Template metaprogram frameworks can use `static_assert` for sig-
naling bad usage of their components in a more user-friendly way than failing the compilation in a deeply nested template instantiation.

Together with type traits [103] static assertion is a convenient tool for adding type constraints to templates, which is a common feature of generic constructs (available in Java, C#, Scala, Eiffel, Ada etc.) [104, 105, 106]. Note that with static assertion these type constraints are not limited to is-a relationship checks [102].

**Metafunction** Functors are objects that can perform a given task. They have several inputs, and one output, the computation, or operation can be invoked through a call to well defined method, to say \texttt{Call} or most typically \texttt{operator()}. Functors are often used in combination with containers, where the actual compare method is factored out to be an external parameter in the form of a compare functor.

Metafunctions are the analogue of functors in the compile time world. They are template classes, where the parameters are the inputs of the functor, and the result is a compile time evaluated public member of the class. The result can be a type in the form of a \texttt{typedef}, an integral scalar, or even a parameterised type. The latter appears as a nested template class.

**Data structures** Data structures are fundamental in most programming environment. Template metaprograms are not an exception. Here the basic data types are constants, types, generic types. If we want to handle some kind of sequence of these as one entity, we have to somehow wrap the elements into a single compile time structure.

A template class with 5 integer parameters can be treated as a fixed 5 length array of integers (see Listing A.7). This approach works, but has two problems: it requires too much work to make the specialisations, and it is still not generic enough. It supports only the integer type and vectors of size 5. The specialisations can be generated automatically with preprocessor metaprogramming techniques, and also variants for different sizes can be implemented in similar manner. Support for arbitrary integral types can be easily added by introducing a type parameter instead of the fixed \texttt{int}. A similar vector data structure can be constructed when we want to store types instead of integral constants. Then the templates have type parameters, and \texttt{typedefs} in the class definition instead of \texttt{static const} values. Though we can implement random indexing operator, and any other useful metafunction, vector-like data structures do not support functional programming very well, because they are not a recursive data type. Type lists have this recursive property, while still supporting much of the possibilities of a linear data type. A type list is actually a pair of a type, and the remaining types. This is the equivalent of the Lisp car and cdr [107]. For a sample type list and indexing implementation
CHAPTER 3. GENERATIVE METAPROGRAMMING


Expression templates Expression templates can represent an expression tree at compile time and allow for deferring actual execution to a later time. By using templates, where the methods are inlined by default, the generated code for performing the actual operations in a later time can remain efficient. The following examples show the strength and usefulness of the approach.

eliminating temporals There are big differences in the cost of memory allocations depending on the place we are allocating from. Stack memory allocation is not more than adjusting the stack pointer, often done once in the beginning of the function [109], only variable length arrays [110] of C and calls to `alloca` [111] account for extra stack allocations. Heap allocations are much more expensive, they operate on a slower data structure, and often involve calls to the operating system. This means heap operations are to be avoided or reduced if speed is an important concern. Sometimes the same data structure is modified multiple times, involving many heap operations, until it reaches its final state. With expression templates [112] many of the heap operations can be exchanged to stack pointer adjustments thus achieving higher performance. In [113], for example, a fast, non-intrusive string concatenation implementation is presented. The solution is based on allocating the final string and directly copying the corresponding substrings into their final location instead of creating numerous intermediate string objects during the evaluation of a complex concatenation expression.

Boost.Bind [114] This package makes creation of functors much easier. The `lambda` [114] library introduces \_1, \_2, \ldots to represent identical projections of the corresponding parameters of the functor being created. That means \_1 is a functor returning its sole argument. Any C++ operator performed on these functor objects produces such a functor, that composes the operand functors and the used C++ operator. \_1 + \_2 is, for example, a functor that adds its two operands. Listing A.8 demonstrates the usage of `bind`. Closures [115, 116] in C++11 fulfil the same demands, and make `bind` much less directly used, other than in legacy code, or where compliance to the earlier C++ standard is required.

Domain specific languages (DSL) [117] are special purpose languages, they make description of domain specific objects very comfortable. A general purpose language cannot be so easy to use for such descriptions as a DSL. Embedded (also called internal) DSLs (EDSL or DSEL) [118, 119] are such DSLs that appear in another (host) language, use host language tokens and form
valid sentences in the host language, but still serving as a special purpose
DSL. Object relational mappings (ORM) [120, 121] are, for example, DSELS
where databases can be accessed just as easily as if they were a fundamental part of the host language. Boost.Spirit [122] is an EDSL for describing
regular grammars embedded in C++. The definition of production rules
is specified by expressions, where C++ operators represent concatenation
(•), iteration (unary * or +) or alternatives (|). Moreover, rules can be an-
notated with actions to be performed when those specific substitutions are
made, thus creating an attribute grammar [123]. The framework generates
the corresponding parser in compile time, that can be used in the application
for parsing sentences of the given language. Boost.Xpressive [122] is an ESDL
similar to Spirit for regular expressions. The metaparse library [124, 125, 126]
not only allows the specification and generation of the grammar at compile
time, but also the generated parser is a template metaprogram. This gener-
ated metaprogram then can be used to create compile time objects like types
or values.

**CRTP** is the acronym of curiously recurrent template pattern, first described
by Coplien [127]. Surprisingly often an optimal solution of a problem is a tem-
plate base class, where the actual type of its derived class is passed as template
parameter: class X : public Base<X>.

This technique is a way to implement static polymorphism. Static polymor-
phism branches execution in compile time, having certain operations implemented
in different classes with the same signature. Specifying the exact type to be used
for these operations decides which implementation is used. Comparator functors
of STL algorithms or policy classes are examples of static polymorphism. With
CRTP the base class can leave the implementation of some methods to the de-
rived class. From this point of view it is the compile time analogue of the template
method pattern [128].

Another application of CRTP is storing separate data for each type in a given
context. As every instantiation of a class template is a separate class, its static
members are an appropriate place to store type specific data. A classic example
for this is a mixin base class that implements object counting, see Listing A.9.

**Generic design pattern implementation** Design patterns [128] form a higher
abstraction level on top of direct language elements. Patterns can help developers
to communicate their design decisions. Most of these patterns rely on object-
oriented concepts, they are applicable in most (if not all) OO languages. TMP
can improve design patterns in different ways. The implementation of the patterns
sometimes requires mechanic coding like implementing the same method in all
CHAPTER 3. GENERATIVE METAPROGRAMMING

derived classes, or repeating signatures because of forwarding. With TMP and/or PPMP these tasks can be automated both for convenience and for consistence.

Apart from automation, with TMP the patterns can be generalised. Alexandrescu demonstrates how the patterns can be encapsulated into TMP constructs to be applied easily at the place of their use [129]. We can find generic implementations for the singleton, visitor and abstract factory patterns and possible approaches for tuple, smart pointer and multimethod. Since then tuple is part of the new C++ standard [130].

Moreover, some of the design patterns can be interpreted in TMP by changing run time polymorphism to static polymorphism. Template method, adapter are such, just to mention a few. A new language element, concepts [131], have been cut out [132, 133] from the latest official standard. Its integration into the language was not final and the committee wanted to release the new standard as soon as possible. With concepts the interface of template constructs can be declared. With concept maps [134] the mapping between similar interfaces can be specified. Concept maps are actually a compile time application of the adapter pattern, supported by a direct language element.

3.2.2 Evaluation

Template metaprogramming is an effective approach to improve performance and code consistency. It can also make otherwise tedious work, such as the proper coding of a design pattern much easier. Template metaprograms use standard language elements, therefore they can integrate well with other parts of the code. Generic construct can be implemented while remaining type safe and keep the possibilities of optimisation. We have iterated types and a Turing complete language that is rather functional than imperative. Through template specialisation most generic solutions can be easily customised for special needs.

As the examples show TMP is used effectively for many different problems. There are, however, some drawbacks of TMP as well. Heavy use of template constructs easily result in long compilation time and/or big executables (the latter effect is called code bloat). Another drawback is the poor readability of template metaprograms. This comes from historical reasons, that templates had not originally designed to form an embedded language inside C++, but only to support generalisation. Partly due to its poor readability, and partly because functional programming in TMP needs different thinking than traditional C++, the learning curve of TMP is quite long.

The last major problem I would like to highlight is the lack of proper tool support. As with preprocessor metaprogramming (see 3.1), template metaprogramming is not the scope of even the most modern development environments. Template processing is treated as an internal mechanism of the C++ compiler.
Diagnostic functionalities are restricted to compilation error messages, where the chain of template instantiations are listed. We cannot step through the template instantiation process with the same convenience as with our run time programs. My theses focus on filling this gap providing ways to instrument, debug, profile and visualise template metaprograms.
Chapter 4

Template metaprogram instrumentation

This thesis chapter is about instrumenting template metaprograms. Section 4.1 describes instrumentation in general and the different implementation approaches. In Section 4.2 we can find the motivation for TMP instrumentation. Then the basic idea behind the template metaprogram instrumentation technique is explained in Section 4.3. The following section (4.4) contains the exact implementation details. The limitations both of the approach fundamentally and the implementation are gathered in Section 4.5. The contribution section 4.7 summarises the achievement and phrase the thesis itself. Finally related work is presented in Section 4.6.

4.1 Instrumentation in general

Instrumentation in engineering means the monitoring, measurement and control of a process [135]. In computer science these measurements are typically about non-functional metrics, like memory footprint or leaks, execution time, CPU load etc. An application or a service has numerous functional and non-functional aspects to keep in mind when performing changes in the system. The first step for preserving or improving the quality of a system is measurement. Though we have this natural demand for being able to measure these values, it is far from being natural that we can do it having an optimised executable. It can easily happen that some information, like total memory used by the application, for example, is available at OS level (we can use OS API to retrieve them), but the important details we would need for making improvements are missing. Much information is simply lost during compilation. Functions are inlined, eliminated or merged during optimisation, for example. The type of the allocated blocks are concepts that exist only at compile time, which is the exact topic of Thesis 5.
4.1. INSTRUMENTATION IN GENERAL

Here measurement has a broader sense, it covers any kind of information retrieval about the system, not only quantitative measures. A debugger is a diagnostic tool, we can confirm good behaviour or pinpoint bugs by breaking execution at certain points and examining variables. Though this test process does not use any quantitative measure, it still can be part of the quality measurement of the system.

The most typical instrumentation technique is adding hooks at the entry points of certain function calls. By capturing calls to allocation and deallocation routines, for example, a memory usage analyser can be implemented (see 7.4.1). Depending on the place in the toolchain there are different solutions for instrumentation.

Manual at source code level One of the simplest approaches is just explicitly calling the hook functions from the appropriate places, practically surrounded with some \#if(def) condition to enable them only in diagnostic builds. This is a good solution if the number of hook points is limited. When the diagnostic is about speed, an automatic instrumentation that inserts extra calls at each function call can ruin the precision of the measurement. While these diagnostic hook actions are typically fast and take negligible time compared to bigger operations, if they are added also to small functions, the measured data will not be reliable [136]. Even if the added operations are as simple as incrementing a global variable, the change in the possible cache optimisations can be substantial [137]. Though it may mean significant effort to add the hooks to each important function, it can still make less harm than an optimisation that is useless because targets wrong functions.

Automatic at source code level When its side effects does not render its application useless, automatic instrumentation is a good choice. It ensures consistency, completeness, and convenience. A way to implement automatic instrumentation is to parse the source code looking for certain patterns, and modify them to call or contain the given hook actions.

In aspect oriented programming [138] there is an extra, so called weaving step, that transforms the source code to include given fragments to certain positions specified usually with regular expressions. As a direct tool for implementing cross-cutting concerns [139], aspect weaver is viable alternative to use for instrumentation where it is available.

Compiler generated Many compilers have built-in support for the most typical instrumentation use case, that is adding extra behaviour at the beginning and/or at the end of each function. When generating position independent code [140] this option may be disabled. Implementation details may differ in whether the hooks are called from within the called function or at the call site. Some compilers leave
this choice to the user. Direct compiler support, if its functionality matches our needs, is very convenient. We only have to implement the two hook functions, see Table 7.1.

**Link time** The connection between user code and the called external functions are finalised in link time in case of static linking. We can add profiler code by generating decorated [141] versions of the functions to instrument. The decorated versions call the original functions, but also perform additional operations to support profiling. We should instruct the linker to prefer our library when resolving references to the instrumented functions.

**On the binary** The source code of the instrumentation target is not always available. If a 3rd party component is received as a compiled binary, for example, and we want to extend our diagnostic onto it, we cannot reach for the source code. Even if we have the source code, a full rebuild whenever we enable or disable diagnostics can be too cumbersome. There are important details that are lost during compilation and optimisation [142], therefore the lack of source code limits our possibilities. In some cases, however, we have all the necessary information even in the binary form. OS API calls cannot be inlined, they can be easily identified in the compiled executable. Of course this is not true if the toolchain contains anti-piracy transformations. These transformations should be turned off for the time of diagnostics. Compiled into the binary, or apart from it, we can have debug information, which further helps identifying the necessary hook points. Similarly to the source code, debug information is not always available. Once we have the hook points in the binary, the compiled code can be extended with hook actions. One approach (called single-instruction overwriting [143]) is to change the assembly instructions of each hook point to a jump [144] to a trampoline [145]. The trampoline executes the hook action code, then performs the original operations at the call site that have been overwritten with the jump instruction. It is important to save and restore registers to ensure they are the same as without the hook action code. VSInstr [146] is a tool for post compilation instrumentation of executable code.

**Byte code** Some languages (most notable example is Java) have a standardised virtual machine (VM) [147] as their compilation target. The VM executes byte code [147]. There are examples of byte code level automatic instrumentation techniques as well [148, 149]. Byte code level instrumentation is combination of source code level and on binary instrumentation.
4.2. MOTIVATION FOR TEMPLATE METAPROGRAM INSTRUMENTATION

**Load time** Load time instrumentation links in profiling code when the application is loaded to memory. Instead of directly loading the dynamic library containing the functions to instrument, a decorated version is loaded that adds profiling code [150]. This is a variant of link time instrumentation. Some libraries contain built-in support for enabling profiler functionalities through environment variables [151]. In case of hybrid [24] or just in time [152] compilation model instrumentation can take place when the byte code is interpreted or translated to machine code. This is actually compile time instrumentation, but in this case the final stage of compilation happens in load time. An example of this technique is the JBoss Pojo Cache [153] that enables load time instrumentation of Java programs [154].

**Run time** Run time instrumentation is much like instrumenting the compiled binary, but it is performed on the running binary in memory instead of modifying the original executable. The Dyninst API [145] is a C++ class library for this type of instrumentation.

If the purpose is instrumenting calls to OS API functions, we can add OS API hooks [155] to the measured process.

Note that instrumentation is not the only way for getting meaningful data out of the execution of an application. Sampling profilers, for example, create snapshots of the call stacks of threads, and other attributes like total used memory, and create a probabilistic report about run time performance. The advantage of this method is its low intrusivity. Disadvantages are that only a limited set of information can be extracted externally and we only have photos of execution instead of data for each call.

4.2 Motivation for template metaprogram instrumentation

As we saw in 3.2 template metaprogramming, though is embedded in C++, has much in common with being a separate programming language. As such, the concepts of more traditional programming can be mapped to it [16]. This includes the need for good diagnostic tools. If we need any type (time, memory, etc.) of profilers for non-TMP programs, we definitely need them for TMP as well [156]. This need is a direct consequence of having large scale template metaprograms. Once they grow to be non-trivial, diagnostics have the same importance as in any other programming paradigm [47]. Having the mentioned mapping between ordinary programming and TMP, we can extend this mapping to cover diagnostic
methods as well. If we were able to somehow instrument template metaprograms, similarly to compiled ordinary programs, and map elements of TMP to their run time equivalent, then we could probably reuse all the existing toolsets for debugging and apply them on our template metaprograms. It turns out that the analogy works very well; the upcoming chapters describe the details.

4.3 The idea

Similarly to profiling in other areas, in template metaprogramming we do not have much information available about the run. In fact, in case of TMP the environment was not prepared to support diagnostics in design time.

It is clear that we should somehow get information out of the compiler about the template instantiations during compilation. It would be nice to do this information retrieval in a platform independent way, which is a design principle of the language itself, after all. Unruh’s prime number generator [40, 41] was a key contributor to the idea. That template metaprogram prints prime numbers in the form of warnings at compile time. This means that we have a method for outputting information from within a template metaprogram without stopping its execution.

What functions are in ordinary programs are template classes or functions in template metaprograms [16]. Therefore the equivalent of a standard instrumentation should add hooks to each template class and each template function. Templates instantiate other templates just like functions call other functions. To be able to follow nested instantiations, similarly to functions, we need hooks both at the beginning and at the end of template classes and functions.

By combining Unruh’s artificial warning generator and the standard instrumentation model, we can get to a working instrumentation method for template metaprograms, see Figure 4.1. In the resulting trace file we have a chronological list of the following sorts of instantiation events:

![Figure 4.1: C++ template metaprogram instrumentation workflow.](image)

46
4.4. IMPLEMENTATION

- beginning of template instantiation
- end of template instantiation
- original warning or error

From the possibly nested begin and end pairs the actual instantiation stack can be reconstructed for the whole process. The preservation of original warnings and errors in the trace file is optional, but recommended. On one hand it can explain otherwise strange situations, like when the end pairs of the already begun instantiations are missing due to an error. On the other hand the toolchain can be simplified if the trace file contains every information that needs to be presented to the user regarding the compilation.

After reading about the different instrumentation approaches in Section 4.1, the question arises which category this method belongs to. The duality of template metaprograms (either simply being parts of the C++ source code or programs interpreted/run by the compiler) makes the answer ambiguous. We can treat the method a source code level instrumentation, because the instrumentation takes place on the source code, not on the compiled binary, after all. However, from the template metaprogram point of view, the metaprogram is actually running when the host C++ code is being compiled. In that sense the compiler is the execution environment, the virtual machine that interprets/runs the code. Viewing from this side, the described instrumentation is closer to the on binary form, even though the executed program is not actually a binary.

4.4 Implementation

This section presents the implementation details of the Templight [4] system. It is a C++ template debugging framework that follows the platform independent philosophy of the language (C++ inherited this philosophy from C [157]). Since the method is based on parsing compiler warnings, apart from platform independence compiler independence is also a concern. First let us see the whole control flow of getting a trace file out of the original source code, see Figure 4.2.

Boxes represent operational elements while unframed texts represent some kind of data. Dashed arrow means optional connection.

4.4.1 Preprocessor

It is not uncommon in template metaprogramming that preprocessor metaprogramming techniques are used to generate different parts of the code [158]. While a compilation unit is specified with a single file name, it can make the compiler have to open further files using \#include directives. Performing preprocessing before any effective analysis simplifies the model to a single document process where
all these external references are already resolved. Using exactly the same preprocessing step as in standard compilation is very important. By using a different preprocessor engine, or the same with different parameters we risk the reliability
of our diagnostic tool.

There is one setting we should add if it is not already enabled: generation of \#line directives. The \_\_FILE\_ and \_\_LINE\_ built-in macros can be used in the code to refer to original file and line positions respectively. Except for those macros in the preprocessed code we cannot distinguish parts of the starting compilation unit from those of \#included other files. For our diagnostic purposes this is not favorable, referring to line numbers in the preprocessed version of the source code during diagnostics is not user friendly enough, the user expects the same behaviour as with the compiler that refers to positions in the original source code. Fortunately preprocessors can be configured to add \#line directives to their output code that identify the original places of the code fragments. See Listing B.3 for an example.

4.4.2 \#line filter

At this point we have a single preprocessed code interlaced with original source code locations. The \#line filter separates the repositioning information from the actual source code. The two result files are one that contains the file and line positions in the form of \#line lines and the source code that does not yet contain any \#line directive. Note that if the original compilation parameters turned \#line generation on, then this step should not remove them. This branching can be easily added to the process.

The purpose of the file/line mapping data is to be able to reconstruct original source code location given a line number in the preprocessed code. For this, it is important not to lose information when extracting the \#line lines. My approach is to prepend each line by its line number in the input file, see Listing B.4. Given this position-conserving \#line extract and the \#line stripped preprocessed code, the interlaced preprocessed code can be fully reconstructed. The position adjust step (described in Subsection 4.4.7) late in the control flow will utilise the mapping created here, while the stripped source code is passed to the annotator and to the instrumentator.

4.4.3 Annotator

The task of annotator is to identify template constructs in the code with every important detail so that the instrumentator can focus only on inserting code fragments. Conceptually it is like highlighting some portions of a text or when proofreaders add side notes to a document. Template definitions can be nested [159], to be able to reflect their structure the output format of the annotator was chosen to be XML. The annotation file contains the exact beginning and end position of
the template along with additional information. It is a trade-off how much information is retrieved here from the source code and in the warning parser phase. The name of the template for example can be easily extracted from the warning message, so the non-trivial task of determining it from the source code can be skipped. More importantly hints for instrumentation points have to be specified accurately. Besides the total boundaries of the template definition, positions of the curly braces are also vital.

To identify template constructs the annotator should parse C++ code to an extent. With the aim to create a compiler independent tool this is not an easy task. Compilers often introduce non-standard extensions which can easily render a standard-compliant parser useless in practice. There are good C++ compiler front ends available [160, 161, 162], but they either have limitations or are too heavyweight. Instead of using one of them, the annotator of Templight uses regular expressions over the C++ tokens. It is a very low level method, with its own limitations, but at the same time it can be implemented easily while remaining compiler and parser independent.

While the recognition of a template definition, be it a class, a (member) function, or a specialisation of them, is helped by their special syntax, this cannot be stated to their end. The end of many C++ constructs is simply a \}. To tell the difference, the parser of annotator should carefully follow each occurrence of \{ and \}. It should keep track of the scope depth of the template definitions (keep in mind that they can be nested), and detect their ends when a \} is met at those specific depths. For diagnostic purposes all detected scopes are also added to the annotation file.

To get C++ tokens from the already preprocessed source I used a Boost library. Boost.Wave [163] is an open source preprocessor engine written in C++. Wave transforms the input character stream containing the source to a stream of C++ tokens. The annotator has to detect the templates in this token stream. For this I chose to use Boost.Spirit which is a parser generator engine. Spirit is generic enough to cope with an input stream of tokens instead of characters.

Figure B.1 shows some of the recognition patterns used in the annotator. For the whole grammer in its Spirit DSEL form refer to Listing B.1. These patterns recognise most template constructs reliably. For the details of their limitations see Section 4.5.

Listing 4.1 and 4.2 show a template class definition and the corresponding annotation as it appears in XML. For a more complete example refer to Chapter B.3.

```xml
<?xml version="1.0" standalone="yes"?>
<!-- [annotation] generated by Templight -->
<FileAnnotation
    beginpos = "test6.cpp|1|1"
```
4.4. IMPLEMENTATION

template <class T, class U>
class Conversion
{
typedef char Small;
class Big { char dummy[2]; }; static Small Test(U); static Big Test(...); static T MakeT();
public:
enum { exists = sizeof(Test(MakeT())) == sizeof(Small) }; enum { sameType = false }; }

Listing 4.1: Example code for demonstrating template class annotation.

Listing 4.2: Annotation result of the code in Listing 4.1.

It is advisable to add a unique identifier to each annotation entry. If they are present in the generated warnings, the warning parser can easily lookup the corresponding annotation entry where additional information can be found. This indirect link by id is not necessary if the warning parser can extract every important
CHAPTER 4. TEMPLATE METAPROGRAM INSTRUMENTATION

```cpp
template<int i>
struct Factorial
{
    enum { value = Factorial<i-1>::value);
};
```

Listing 4.3: Sample code before instrumentation. Each \* represents an instrumentation point.

```cpp
template<int i>
struct Factorial
{
    struct _TEMPLIGHT_0s
    {
        int a;
    };
    enum { _TEMPLIGHT_0 = Templight::ReportTemplateBegin<_TEMPLIGHT_0s, &_TEMPLIGHT_0s::a>::Value };  
    enum { value = Factorial<i-1>::value };  
    struct _TEMPLIGHT_1s
    {
        int a;
    };
    enum { _TEMPLIGHT_1 = Templight::ReportTemplateEnd<_TEMPLIGHT_1s, &_TEMPLIGHT_1s::a>::Value }
};
```

Listing 4.4: Sample code after instrumentation.

detail from the warnings themselves. The dashed connection between annotation and warning parser denotes this optionality in Figure 4.2.

4.4.4 Instrumentator

Based on the annotation instrumentator inserts such code fragments into the source code that trigger compilation warnings. This step is compiler dependent, for each supported compiler we have to find the exact way of triggering such a warning that contains enough information for identifying the given template construct. Fortunately many compilers automatically list the whole instantiation chain, or at least the given template instance where an error or warning occurs. If the connection between the instrumentation point and the corresponding annotation entry is important, the id of the annotation entry can be incorporated into some class name or other identifier in the warning generator code so that it is present in the warning text. Listing 4.3 and 4.4 show a sample code before and after instrumentation respectively.

Sometimes the added codes need helper functions/classes that should be de-
declared beforehand, in an outer scope. I call this the preamble that is a fixed code added to the beginning of each instrumented source file, containing all the infrastructure possibly used in the inserted fragments.

Because of changes done in the source code, similarly to preprocessing we have to make sure that a location reference to the modified code can be mapped back to the input source. Apart from the patched source code, the other output of instrumentor is a line mapping that lists how many extra lines have been added at each specific location. As Listing 4.5 shows, the format is very simple: the original line number is in the first column, and the number of added lines at that position is in the second. See also Listing B.7.

When looking for a proper warning trigger code for a compiler, the following requirements should be considered:

- Should trigger a warning, but not an error. Errors can stop compilation either immediately or after reaching a limit.
- The generated warning should contain all information we require. Either an id that refers to the annotation entry, or the name of the template instance, maybe both.
- Should work in each possible context where it is inserted. We can have different fragments to use in case of template functions, template member functions and template classes. Nested templates can behave differently so they deserve explicit testing.
- Not to affect template processing. If the inserted code triggers further template instantiations, the instrumented version will not reflect the original compilation process.
- Not to interfere with other code or other warnings. If the added code introduces identifiers, they should be extremal to avoid name clash with the original code. It is worth adding a unique suffix to avoid clash with other instrumented fragments in the same scope.
4.4.5 Compiler

In this step the original compilation command with the same parameters has to be performed on the modified source code. This compilation should be as close to the original as possible. There are two differences: preprocessing has already been done as a separate step, and extra code fragments have been added.

The textual output of this compilation phase is passed on to the warning parser for information retrieval.

4.4.6 Warning parser

There is a strong connection between the warning parser (WP) and the instrumentator. WP has to recognise the warnings triggered by the code inserted in the instrumentation phase. WP should be prepared to encounter not only warnings triggered by the instrumented code fragments, but messages (#pragma message citegcc), warnings or errors originally present in the compilation output. It is an important aspect in the construction of the trigger code to make its compilation output stand out from the normal output. This is achieved by using extremely improbable identifier names: they all have a _T_E_M_P_L_I_G_H_T___ prefix.

In the current Templight implementation WP extracts all the necessary information from the warnings themselves and the annotation information is used only by the instrumentator. As an alternative implementation WP could only extract an annotation id and refer to the annotation for additional information generated in the annotation phase. Either way, WP interprets the compiler output and transforms it into a sequence of trace events.

Table 4.1 summarises the different types of trace events and their properties. The output is a flat structured XML file that lists the events in their chronological order. See Listing 4.6 for an example.

<table>
<thead>
<tr>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>template instantiation begin</td>
<td>name, actual parameters</td>
</tr>
<tr>
<td>template instantiation end</td>
<td>-</td>
</tr>
<tr>
<td>typedef</td>
<td>identifier, type</td>
</tr>
<tr>
<td>message</td>
<td>message text</td>
</tr>
<tr>
<td>error or warning of original code</td>
<td>corresponding compiler output</td>
</tr>
<tr>
<td>all</td>
<td>code position</td>
</tr>
<tr>
<td>all</td>
<td>optional: timestamp</td>
</tr>
</tbody>
</table>

Table 4.1: Types of trace events and their properties.
4.4. IMPLEMENTATION

...<TemplateBegin>
  <Position position="test.cpp|9|1" />  
  <Context context="Factorial<5" /> 
</TemplateBegin>
<TemplateBegin>
  <Position position="test.cpp|9|1" />  
  <Context context="Factorial<4" /> 
</TemplateBegin>
...

Listing 4.6: Sample trace file fragment.

4.4.7 Position adjust

At this point we have a trace file that describes the template instantiation process in the form of events. Source code location references in this trace file point to the instrumented source, which differs from the original because of preprocessing and instrumentation. In this step the locations are remapped to their equivalent in the original source code based on the line mapping and file/line mapping information saved in the corresponding transformation steps.

Let \( p_i \) denote the position (line number) of the \( i \)th patch in the preprocessed source, and \( l_i \) the length (number of lines) in that patch. The line mapping file is a list of the \((p_i, l_i)\) pairs. From these values we can calculate where the patches start in the resulting file:

\[
s(i) := p_i + \sum_{1 \leq k \leq i} l_k \quad (1 \leq i \leq N) \tag{4.1}
\]

To make lookup easier we also introduce a virtual patch literally at infinity:

\[
p_{N+1} := \text{INT\_MAX} - 1  \\
l_{N+1} := 1
\]

which yields a valid value for \( s_{N+1}. \) \( s \) is strictly increasing, therefore we can find the index of the nearest next patch in logarithmic time for any \( r \) line number in the resulting file:

\[
n(r) := \min \{ i | s(i) > r \}
\]

\( 1 \leq n(r) \leq N + 1 \) for each valid \( r. \) There are two cases: a) \( r \) is part of patch \( n(r) - 1 \) or b) it is a line between patches \( n(r) - 1 \) and \( n(r). \) The line number in

55
the preprocessed file before instrumentation is calculated as follows:

\[
b(r) := \begin{cases} 
n_{n(r) - 1} & \text{if } n(r) \geq 2 \land r < s(n(r) - 1) + l_k, \\
\sum_{1 \leq k \leq N} l_k & \text{otherwise.}
\end{cases}
\]

The second branch can be rewritten to

\[
r - \sum_{1 \leq k \leq N} l_k = r - (s(n(r) - 1) - p_{n(r) - 1})
\]

by using 4.1 with \(i = n(r) - 1\) and that

\[
\sum_{1 \leq k \leq N} l_k = \sum_{1 \leq k \leq N} l_k
\]

All in all, with an \(O(N)\) time precalculation of \(s\) that can be done along with reading the input, \(b(r)\) can be computed in \(O(\log N)\) time for any valid \(r\).

The mapping from the preprocessed line numbers back to the original file and line location is based on the file/line mapping file, which is a list of \((p_i, l_i, f_i)\) triplets. These values were obtained from the \#line decorated preprocessed file:

- \(p_i\) := position (line number) of the \#line directive
- \(l_i\) := original line number of the next \((p_i + 1)\)th line \((1 \leq i \leq M)\)
- \(f_i\) := original file path of the next line

Given a \(b\) line number in the preprocessed file before instrumentation, we have to find the index of the last \#line directive in effect:

\[
e(b) := \max \{i | p_i < b\}
\]

\(p_i\) is strictly increasing, therefore \(e(b)\) can be found in logarithmic time. To make \(0 \leq e(b) \leq M\) for every \(b\), we introduce

\[
p_0 := 0
\]
\[
l_0 := 1
\]
\[
f_0 := \text{file path of the starting compilation unit}
\]

The line number and file path in the original source then can be determined easily:

\[
l^o(b) := l_{e(b)} + b - p_{e(b)} - 1
\]
\[
f^o(b) := f_{e(b)}
\]
Similarly to the line backmapping, with an \( O(N) \) precalculation (storage) of \( p \) the file/line backmapping can be done in \( O(\log M) \).

The calculations above are valid only if the \#line directives have not been filtered out from the preprocessed file at the \#line filter step. If they have, the figures are a bit different. Let

\[ p'_i := p_i - i \]

denote the position (line number) of the first line where the \( i \)-th \#line directive is in effect. \( p'_i \) is the counterpart of \( p_i \) in the filtered file. Exactly \( i \) \#line lines have been cut out from the preprocessed file that \( p_i \) refers to, \( b \) should be looked up in the \( p' \) table. Note the \( \leq \) instead of \( < \) and that the \(-1\) correction is not needed in \( l^0(b) \):

\[
\begin{align*}
    e(b) &:= \max\{i | p'_i \leq b\} \\
    l^0(b) &:= l_{e(b)} + b - p_{e(b)}
\end{align*}
\]

All the other figures and the complexity remain the same.

After backmapping the positions of the input trace file first based on the line mapping, then on the file/line mapping information we finally get a trace file that can be directly used by the applications. In the resulting trace file the location references point to the original source code, even directly to the starting compilation unit or to other files that were part of processing via one or more \#include directives.

## 4.5 Limitations

The template metaprogram instrumentation technique described in this chapter has some limitations that have to be taken into account during its usage. Besides the problematic cases this section also enumerates the compiler dependent parts of the procedure, and what explicit steps are needed to extend the framework with the support of a not yet covered compilation environment.

### 4.5.1 Pattern recognition difficulties

The annotator uses regular expressions to identify template constructs in the preprocessed source code. The expressiveness of regular expressions is not enough to cover all cases of C++ template definitions. The grammar of C++ is context sensitive [83], meaning that it cannot be parsed even with an LALR(1) parser [164], let alone regular expressions.

Some tokens have different meaning depending on the actual context, the angle brackets \((<,>)\) are a notable example. Listing 4.7 shows a code that has different
CHAPTER 4. TEMPLATE METAPROGRAM INSTRUMENTATION

Listing 4.7: Context sensitivity in the C++ grammar: two possible, but very different interpretation of the same token sequence depending on an earlier declaration.

Listing 4.8: Template function declaration containing ) token in the argument list.

meaning depending on an earlier declaration. Here < can either be a relational operator or the beginning of the template parameter list. There can be arbitrary number of other tokens between the declaration and the code depending on it, we cannot rely on the declaration and its usage being close to each other. In template metaprogramming enum definition is a way of storing computed integral values [165], a counterpart of typedefs which are used to store types. Consequently properly recognising enumerations by the annotator would definitely make the annotation more useful in describing the structure of the template metaprogram.

The limitation of using regular expressions can be witnessed in the pattern in Figure B.1. It fails to recognise template (member) functions where the argument list contains ) token, see Listing 4.8. Currently these templates are out of sight of the annotator. In a concrete application the recognition patterns can be customised to cover all important cases.

4.5.2 Inheritance

Instead of direct forwarding, public inheritance is often used in template metaprograms to import type or value definitions [47]. It avoids repeating the identifier and
needs less typing. The current implementation of the Templight framework defines instrumentation points at the { and } tokens of the class definition, embracing all member definitions. The list of base classes, however, are before the { token, see Listing 4.9. Figure 4.3 shows the trace events of the Factorial code in its base and variant versions respectively. While the mechanism of the two implementations are the same, Factorial<N> triggers the instantiation of Factorial<N-1>, only the base version reflects this recursive structure, the variant version flattens the call.

<table>
<thead>
<tr>
<th>#undef VARIANT</th>
<th>#define VARIANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial&lt;0&gt; begin,end</td>
<td>Factorial&lt;0&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;3&gt; begin</td>
<td>Multiply&lt;1,1&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;2&gt; begin</td>
<td>Factorial&lt;1&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;1&gt; begin</td>
<td>Multiply&lt;2,1&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;1&gt; end</td>
<td>Factorial&lt;2&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;2&gt; end</td>
<td>Multiply&lt;3,2&gt; begin,end</td>
</tr>
<tr>
<td>Factorial&lt;3&gt; end</td>
<td>Factorial&lt;3&gt; begin,end</td>
</tr>
</tbody>
</table>

Figure 4.3: Instantiation events of the code in Listing 4.9, base and variant versions. begin,end is short for consecutive begin and end events.
sequence. The instrumentation point at the \{ token detects template instantiation only when the base classes have already been processed.

A workaround to this problem is moving the instrumentation point from the \{ token to an extra 0th empty base class.

### 4.5.3 Template processing mechanism

It is important to keep in mind that the full instantiation of a template with certain parameters can happen at most once during compilation. See also memoisation later in 5.3.2. Whenever the same specialisation is used later, the instantiation does not happen again, consequently the trace will not contain new instantiation events for the same specialisation either. When the compiler finds a template definition, it has to parse it slightly, at least following the scope changes to find its end. Some compilers go beyond this, and perform more intense semantic checks. The C++ standard defines the notion of dependent name and non-dependent name. Dependent names are those depending on some of the template parameters. Non-dependent names are resolved during initial parsing, while dependent names are only during instantiation. Because of this two-phase processing of templates the instrumented code can trigger a warning not only at the time of instantiation, but also, or only at initial parsing. Both can mislead the warning parser and result in an imprecise trace file.

### 4.5.4 Compiler (in)dependence

As Figure 4.2 shows the Templight framework consists of original, compiler independent and compiler dependent steps. In most cases when porting the system to a new compilation environment, only the instrumentator and warning parser need major extensions. If the extraction of all the necessary information is not possible via warnings, annotator may also need enhancements to supply more details on the templates of the code.

The current implementation of Templight uses a catch-all warning parser. The same parser code recognises the warnings of different compilers, containing patterns for all supported versions. This approach simplifies the workflow if different compilers are used on the same platform: only the instrumentator should be invoked differently. Though the instrumentator could be implemented in such a way that it adds all the different warning trigger codes of different compilers, it would not be practical. In one hand some compilers may treat the inserted code an error instead of a warning. On the other hand the multiple warnings for the same event would make the proper implementation of the warning parser more difficult.

It is better referring to compilation environments and not simply compilers. A certain compiler can have multiple versions, it can behave differently depending
on the compilation flags, or on the running platform. The notion of compilation environment covers all these subtle details as well. Even if we do not change the compiler totally, only upgrading it to a newer version creates a new compilation environment. Every such change should involve checking if the warning trigger code works as expected and exploited.

The key in adding support for a new compilation environment is finding the proper warning trigger code. Once it is found, writing a parser for it is a simple task. Finding a good warning trigger code needs experimenting. Table 4.2 lists some examples along with samples of the warnings they generate.

<table>
<thead>
<tr>
<th>trigger code</th>
<th>generated warning</th>
</tr>
</thead>
<tbody>
<tr>
<td>// preamble:</td>
<td></td>
</tr>
<tr>
<td>int f(char*);</td>
<td></td>
</tr>
<tr>
<td>// IP:</td>
<td></td>
</tr>
<tr>
<td>static const size_t v = sizeof(f&quot;&quot;));</td>
<td></td>
</tr>
<tr>
<td>warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]</td>
<td></td>
</tr>
<tr>
<td>warning: conversion from string literal to 'char *' is deprecated [-Wdeprecated-writable-strings]</td>
<td></td>
</tr>
<tr>
<td>static const unsigned v = -1;</td>
<td></td>
</tr>
<tr>
<td>warning: negative integer implicitly converted to unsigned type [-Wsign-conversion]</td>
<td></td>
</tr>
<tr>
<td>warning: implicit conversion changes signedness:</td>
<td></td>
</tr>
<tr>
<td>'int' to 'const unsigned int' [-Wsign-conversion]</td>
<td></td>
</tr>
<tr>
<td>static const unsigned v = -1.0;</td>
<td></td>
</tr>
<tr>
<td>warning: overflow in implicit constant conversion [-Woverflow]</td>
<td></td>
</tr>
<tr>
<td>error: in-class initialiser for static data member is not a constant expression</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Warning trigger codes and the generated warnings with g++ 4.8.2 and clang 3.1. Identifier names are simplified, in the real application they are intentionally verbose and strange to make clash unlikely.

### 4.6 Related work

Vandevoorde and Josuttis recommends creating a tracer class that emits run time messages in its operations [166]. When a template is instantiated with the tracer
class, the run time messages show what operations the template performs on its parameter class. The authors also defined archetype as a type specifically designed to test templates against unwanted dependencies on their parameter. The archetype defines only those operations that are included in the interface contract.

Abrahams and Gurtovoy devoted a whole chapter to diagnostics in their book on Boost [47] They use manually inserted warning generator code to inspect the template instantiation process. They also present mpl::print which is a utility template to print a type in form of a warning at compile time.

In 2008 Steven Watanabe implemented a template instantiation profiler based on the same instrumentation technique as of Templight [167]. His recognition patterns do not distinguish templates and non-templates, instrumentation is performed on non-template classes and functions as well. The patterns handle ) tokens in argument lists properly. Deletion of an incomplete type or division by zero code is instrumented depending on the compiler environment. As of 2014 the tool supports gcc and msvc compilers. The output of the tool is a list of lines containing function or class definition, each with an instantiation count. For non-templates the instantiation count is 1. Later this tool was improved by Domagoj Saric to work as a standalone binary [168].

4.7 Contribution

Thesis 2 (Template metaprogram instrumentation). I developed a method for instrumenting C++ template metaprograms. The basic concept of the method is the pattern based recognition of template constructs and the insertion of warning generator code fragments. Based on the method I implemented a prototype framework that outputs a position-correct trace file containing all necessary information to reconstruct the full TMP execution. The framework can be adapted to new compilation environments easily.
Chapter 5

Template metaprogram debugging

In this chapter different methods are discussed for debugging template meta-
programs. Section 5.1 starts with motivation, then the analogy between run time
and TMP is presented from debugging viewpoint in Section 5.2. Two different ap-
proaches are shown for implementing a template debugger in Section 5.3. The first,
more portable method is based on the instrumentation technique of Thesis 2, the
other requires modification of the compiler, but eliminates some of the limitations
of the first method. Section 5.4 shows three applications that use the described
approaches. Related work is presented in Section 5.5, then my contribution is
summarised in Section 5.6.

5.1 Motivation

Considering the role of TMP in today’s modern C++ programming, its tool sup-
port is far below current standards. While we can step through all statements, or
even assembly instructions on each running thread in case of run time programs,
we should fall back on using diagnostic print (mpl::print) statements for debug-
ing TMP. Within the two decades that passed since Unruh’s prime metaprogram,
this lack of proper debugging tools has been more and more obvious, and became
one of the key C++ concerns to solve. Scott Meyers explicitly warns people who
are enthusiastic about TMP techniques that they will have no convenient tool to
figure out what happens if something goes wrong [169]. When asked what tool for
C++ he would choose if he could have only one wish, Stephan T. Lavavej spoke
about an interactive, step through debugger for TMP [170].

Programming, consequently TMP too, is a human activity, and as such, it is
error-prone. Postdelivery maintenance tends to be the vast majority of software
cost, 70-80% proportion is not rare [171]. Maintenance often involves debugging, consequently providing sufficient tools to make it more convenient is essential. Note that in this sense debugging does not necessarily mean only finding the root cause of erroneous program behaviour, but also covers exploratory debugging when existing functionalities are to be added and reverse engineering is needed.

A debugger is a tool that reveals the details of program execution to the programmer. The merit of a debugger is that it shows the real execution. Programmers may already have a mental model of the program and an imagination about its execution. I will refer to this mental view of execution as presumed execution. Table 5.1 shows the relation of presumed and real execution in different debugging scenarios. Debugging operations, like placing breakpoints, step in/over/out

<table>
<thead>
<tr>
<th>purpose</th>
<th>precondition</th>
<th>goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>exploration</td>
<td>presumed = N/A or incomplete</td>
<td>presumed := real</td>
</tr>
<tr>
<td>verification</td>
<td>presumed may differ from real</td>
<td>make sure that presumed = real</td>
</tr>
<tr>
<td>problem solving</td>
<td>presumed ≠ real</td>
<td>find the root cause of difference</td>
</tr>
</tbody>
</table>

Table 5.1: Relation between presumed and real execution in different debugging scenarios.

help the programmer traversing real execution effectively. These operations are invoked by following an intuitive or an explicit debugging strategy, the wolf fence algorithm for example [172]. Print debugging is inferior to interactive debugging because practically each debugging operation requires additional print statements and the accompanying (re)compilation and rerun. The more we can ease the debugging process by providing convenient and powerful tools that reveal the underlying details of program (TMP in our case) execution, the more we can reduce the often 70-80% proportion of software maintenance cost.

5.2 Debugging run time vs. TMP

As we have seen in Section 3.2 templates form a functional programming language embedded in C++. Table 5.2 shows how specific language elements appear in a run time functional program and in C++ template metaprograms. C++ programmers are used to debuggers for run time procedural/object oriented programs. Based on the mapping between language elements and C++ TMP constructs we can determine the equivalents of run time debugger functionalities for template metaprograms.
In case of run time programs the fundamental event during execution is a function call. Function calls are directly supported even in the assembly language on most platforms. On the other hand it also appears in high level contexts such as a method call, which is, having an implicit `this` parameter, a function call after all. Some object oriented languages (Smalltalk, Objective C) prefer message passing terminology, but message passing is technically the same as a method call in most cases.

Many functionalities of run time debugging are in direct relation to function calls. We can see the call stack, which is the actual list of functions calling each other. The elements of the call stack are called frames. A frame is more than a function in that it contains contextual information: what values the registers contain, especially the actual value of the stack pointer and the instruction pointer. By changing frames we can inspect local variables (that are stored in registers or on the stack) of the given calling context, which is another function.

The equivalent of a function call in C++ template metaprograms is template instantiation. Template instantiation is a fundamental event, plays a central role in TMP execution. Along this analogy we can talk about instantiation stack,
which is the compile time analogue of the run time call stack, and we can map all the other notions of run time execution which are based on function calls. Table 5.3 summarises how specific elements of run time execution relate to template metaprogram execution.

<table>
<thead>
<tr>
<th>Run time execution</th>
<th>C++ TMP execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>function call</td>
<td>template instantiation</td>
</tr>
<tr>
<td>call stack</td>
<td>instantiation stack</td>
</tr>
<tr>
<td>call site</td>
<td>reference to another template instance</td>
</tr>
<tr>
<td>frame</td>
<td>actual template instance with values of local symbols, including template parameters</td>
</tr>
<tr>
<td>function parameter</td>
<td>template parameter</td>
</tr>
<tr>
<td>return value</td>
<td>the referenced nested type or value</td>
</tr>
<tr>
<td>instruction pointer</td>
<td>part of source code being semantically parsed</td>
</tr>
<tr>
<td>local variable</td>
<td>type or constant value in the given context</td>
</tr>
<tr>
<td>function body</td>
<td>template definition</td>
</tr>
</tbody>
</table>

Table 5.3: How specific run time execution elements relate to C++ template metaprogram execution.

Now that we have a mapping between runtime and TMP execution, we can also extend run time debugging operations to C++ template metaprograms. Table 5.4 shows how debugging operations can be applied on C++ template metaprograms based on the analogy.

<table>
<thead>
<tr>
<th>Debug operation</th>
<th>C++ TMP debug operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>place breakpoint at a statement</td>
<td>place breakpoint at source code location</td>
</tr>
<tr>
<td>step in/over/out function call</td>
<td>step in/over/out instantiation</td>
</tr>
<tr>
<td>inspect value of variable</td>
<td>inspect type or constant</td>
</tr>
<tr>
<td>continue execution</td>
<td>continue compilation</td>
</tr>
</tbody>
</table>

Table 5.4: How specific run time execution elements relate to C++ template metaprogram execution.
There are two properties of TMP that make their execution more static than that of a run time program.

1. The lack of assignment that comes from the pure functional nature of TMP. Referential transparency is a key property of pure functional languages, meaning that a certain expression evaluates to the same value regardless of the place and time of its evaluation within program execution. (The official definition is phrased a bit differently [173]: program behaviour does not change if an expression gets replaced by its value.) Once a TMP type or constant has been initialised, its value remains the same until the end of compilation. Similarly, instantiation happens at most once. For any template specialisation, therefore, there is a single point in the run where its instantiation happened. The same is true for constant and type initialisations. Moreover, all template instantiations and constant value or type definitions should fit into compiler memory, because they should be available for the remaining part of the compilation. In a run time program, memory is usually often overwritten, and complex dynamic behaviour makes the creation of an all-including trace file practically impossible. For TMP, however, the compilation process itself shows that a full trace should easily fit into memory, since similar information is kept during compilation.

2. Run time programs may have virtually endless possible runs depending on their input. A template metaprogram runs the same way if the source and the preprocessor defines are unchanged. Sometimes this fully deterministic property becomes a disadvantage, for example when we want a proper seed for our compile time random number generator [174]. Apart from the rare cases when our template metaprogram depends on something dynamic (like \_\_DATE\_\_ or \_\_TIME\_\_), we can assume that the run of our metaprogram is actually a fixed sequence of events.

Instead of always thinking in a local context, as in case of run time debugging, the properties above enables a TMP debugger to deal with the whole run. With that approach new debug operations can be identified, such as navigating back and forth in time. (In the recent years several mainstream debuggers implemented reverse debugging [175]. Execution can be played backwards using previously captured contextual information.) We can have quick time jump operations like going to the point where a specific template instantiation or constant/type initialisation takes place.
CHAPTER 5. TEMPLATE METAPROGRAM DEBUGGING

5.3 Different ways to implement a template debugger

When we are talking about an interactive debugger, interactivity can refer to two separate things. On one hand the user interacts with the debugger by entering commands or controlling a graphical user interface. On the other hand the debugger itself can interact with the debugged program by modifying its internal states: the memory or the registers. Of course this latter is also done by the user’s request. In case of template metaprograms there is not really such as an internal state. Further execution can depend on the result of earlier events, and the ability to change values of some constants or types could make sense for checking what-if scenarios during debugging. Allowing or not such execution-changing interactions differentiate trace based and fully interactive debuggers. Fully interactive debuggers should work closely together with the compiler in order to make changes in the value or type definitions possible. That interactivity also invalidates all the benefit that comes from the static nature of TMP. The purpose of the two methods presented in this section is capturing a full trace of TMP compilation to be used for trace based debugging.

5.3.1 Instrumentation based approach

Chapter 4 presented the Templight framework, which is a tool chain for instrumenting C++ template metaprograms. Templight is able to generate a trace file describing template instantiation events by extracting data about TMP run via warning messages emitted by instrumented code fragments. Source locations in these warnings refer to the instrumented code, but line mapping information is saved during instrumentation to make the necessary position adjustment possible. See 4.4.7 for the details of this adjustment. Positions in the trace file, therefore, accurately identify the place of template events in the original source code.

Advantages One of the biggest advantages of the instrumentation approach is its compilation environment independence. To support a new compilation environment, only the warning trigger code and the corresponding warning message parser code should be implemented, with good chance of being able to reuse already existing ones. In special cases new recognition patterns may be required if the existing instrumentation points are not appropriate for warning generation, or additional information should be extracted in the annotation phase to have more information in the warnings.

By adding this little work, that is only needed when facing a new compilation environment, we end up with a straightforward way of creating traces of TMP
5.3. DIFFERENT WAYS TO IMPLEMENT A TEMPLATE DEBUGGER

runs. The steps can be added to the build system very easily. The trace contains the necessary elements for basic debugging operations. From the template instantiation begin and end events we can reconstruct the instantiation stack all the way through the compilation process. The annotator identifies the source code location of template definitions, therefore our debugger is able to show exact source code locations as the user invokes debugger commands. Standard control flow operations, like step in, through, out, can be implemented. Less typical debugger operations, like direct jump in time, are available by exploiting that we have control over the whole compilation, examining it posteriorly. Original warning and error messages are available, interlaced with template related events, in exact chronological order.

**Limitations**  Regular expressions are not able to properly recognise every possible template construct. This may lead to instantiations not being listed in the trace file. As we have seen in the instrumentation chapter, base class instantiations are not reported to be part of the instantiation of the derived class, resulting in misleading order of trace events.

Instrumentation is very sensitive to language change. Migrating from C++03 to C++11 or C++14 requires the corresponding update of the recognition patterns and also the warning parser even when using the same compiler on the same platform. Not only compiler versions and platforms, but different language versions should also be treated as different compilation environments.

When the project uses different compilation configurations, transferring the actually used compilation parameters to the Templight framework may be non-trivial. While this is not a problem with a make based, or similar build system, it is when configuration settings are stored in the project files of the IDE.

Though the basic debugger operations should work fine, an instrumentation based approach is not capable of monitoring internal actions of the compiler, such as memoisation (see later in 5.3.2), for example.

All in all we can say that instrumentation is a good choice when modifying the compiler is ruled out either because its source code is not possible, or because it does not seem to be worth the effort compared to the demanded debugger features. With relatively few investment we can get an acceptable level of debugging capability. On the other hand it is very important to be aware of the limitations when opting for this method.

### 5.3.2 Modifying the compiler

A more reliable trace generation can be implemented by adding it directly to the compiler. The first step towards a built-in, natively supported template debugger
facility is getting the source code of the compiler. Gcc [176] and clang [177] are open source C++ compilers, their source code is available for download. EDG (Edison Design Group) [178] has a C++ front end, that transforms C++ code to an internal representation that can be used for analysis or code generation. Many code analyser and compiler software use that front end to parse C++ code. For their customers EDG ships the full source code of the compiler front end.

If the source code of the compiler we are using for development is not available, we can decide to fall back on using the instrumentation method, or modify a different compiler and use this other compiler for template debugging. Of course, in this latter case we are not debugging the same compilation process, which is fundamentally wrong. On the other hand template processing has its defined rules in the standard, every compiler should instantiate templates according to the same rules, in a similar manner. We can use a different compiler for debugging, but then we should be prepared for not seeing exactly the same events as with the original compiler. This is a compromise to judge, if better debugger abilities are worth the uncertainty. A hybrid version could be to use the instrumentation method and the modified compiler side by side, and compare their event flow to detect possible differences.

Once we have chosen our compiler to modify, and set up the build environment, we have to find the ideal hook points to catch template instantiations. Table 5.5 shows what source files contain template processing related code in gcc and clang. It is recommended to introduce a new command line flag for enabling trace generation instead of compilation. Doing that helps keeping original functionality safe from our modifications and we can easily skip any processing that is not needed for diagnostics. Code generation can be disabled entirely, for example, which is a major speedup.

At the hook points we have to print the details of the template related event. Source code locations and type names should be printed in a human readable form. We can refer to the diagnostic module of the compiler to find what functions to use for these kinds of output.

Once we could put together the simplest trace file, we can go on improving trace generation by adding additional features. For basic template debugging operations template instantiation begin and end events are sufficient, but adding support for

<table>
<thead>
<tr>
<th>compiler</th>
<th>source path</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>gcc/cp/pt.c</td>
</tr>
<tr>
<td>clang</td>
<td>lib/Sema/SemaTemplateInstantiate.cpp</td>
</tr>
<tr>
<td></td>
<td>lib/Sema/SemaTemplateInstantiateDecl.cpp</td>
</tr>
</tbody>
</table>

Table 5.5: Source files containing template processor codes in gcc and clang.
5.3. DIFFERENT WAYS TO IMPLEMENT A TEMPLATE DEBUGGER

typedef, enum or const literal definitions can make our debugger more powerful.

Having full access to the internal mechanisms of the compiler we can reflect internal processes in the trace file. Template instantiation takes place at the first need of the definition of the given template instance. For all further references of the same instance the already instantiated object is used by the compiler instead of performing instantiation again and again. This caching mechanism is called memoisation. When debugging template metaprograms, we may be interested not only in the first reference of a template instance, that triggered instantiation, but also the subsequent ones which were served by memoisation. Memoised references can be added to the trace file just as simply as instantiations, once we located the place where the memoisation cache is checked for the existing instance.

Advantages  The clear advantage of a built-in trace generator is the very close integration with the compiler. All functionalities of the instrumentation based trace generation can be implemented easily. The tool chain can remain intact, there is no need for extra processing steps (location mapping, annotation, instrumentation, warning extraction, position adjust) to get precise information on template events, and source code locations are also accurate with no extra effort. We can trace practically anything we want regarding template processing, we have all the necessary information at hand. That means we can track not only instantiations, but memoisation as well. Handling enumerations, complex typedefs, function definitions or inheritance is not limited by parsing challenges. Listing C.1 shows a trace file that contains not only instantiation, but also memoisation events. This trace file was generated by a modified version of the clang compiler [14].

Limitations  Applying the compiler modification technique has some requirements that are not always easy to fulfil. First, we either need access to the source code of our compiler or settle for using a different compiler for template debugging with all the risk that imposes. Second, even if we have the full source code, finding all hook points is not an easy task. Identifying the proper hook points often requires deep understanding of both compiler implementation techniques and the subtle details of the language.

Sometimes the implementation of similar language features are scattered in the source code. Making sure that we have added trace code to all important cases is difficult.

By modifying the source code of the compiler, we created a new development branch of it. Migrating our trace generator modifications to a newer compiler version may need extra work depending on how much the affected code changed in the meantime. One strategy to address this problem can be to have our changes incorporated into the main development branch of the compiler. This, however,
CHAPTER 5. TEMPLATE METAPROGRAM DEBUGGING

requires a decent implementation that meets the possibly rigorous coding standard and design requirements [179].

In this section two different approaches were presented to have trace files describing template processing events been generated as an input for the post-TMP-run interactive debugger. The exact format of this trace file is intentionally left open, treated as implementation detail. The suitable format depends on the concrete situation, the implemented features, the preference of the developers, the requirements of the tools we want to process it with.

5.4 Applications

So far we have seen how to generate a trace file that describes TMP processing details. Though it is not impossible to use this kind of trace file directly for analyzing template metaprograms, interactive frontends can make debugging much less cumbersome. In this section three of such frontends are presented. Each uses the same trace.xml format and shows a fundamentally different approach to create a graphical debugger frontend:

1. Reuse an existing debugger, and add TMP debugging as a plugin.
2. Create a debugger framework from scratch, but preserve the structure and operations of a run time debugger.
3. A standalone tool entirely tailored for TMP debugging.

5.4.1 IDE extension

Many integrated development environments (IDE) allow their functionalities to be extended by implementing plugins. In Visual Studio® (VS for short) they are called add-ins. The philosophy behind my Templight add-in is to reproduce run time debugging commands for TMP debugging. By switching Templight mode on, the actually opened source file gets instrumented, and its TMP run is available for debugging with the usual step in/over/out and continue commands while it stops at breakpoints placed inside template definitions. Switching Templight mode off stops TMP debugging and flushes the internally stored trace data. Figure 5.1 shows how the Templight backend is triggered by starting a TMP debugging session. Trace file generation takes place once at the beginning of each TMP debugging session. The add-in then uses the trace to act as if the TMP ran at the time of debugging. This virtual execution pauses either at breakpoints, at end of trace, or as prescribed by the last command Step in stops at any subsequent instantiation. Step over stops at the end of the actual instance or when the next source line is reached in the actual source file. Step over proceeds recursively with
all subsequent instantiations which are originally triggered from the same source code line. Step out stops at the end of the actual instance. The add-in adds two additional windows: instantiation stack and the trace view. Instantiation stack is the TMP equivalent of the run time call stack. Trace view shows the full trace as a sequence of events, highlighting the actual position of virtual execution. Figure 5.2 shows how a typical debugging session with the add-in looks like in the IDE. For breakpoints to work properly the trace files should contain references to the point of instantiation in the instantiation begin events. During virtual execution the source code range of the actual template (topping in the instantiation stack) is processed line by line until its end, or until the place of the next instantiation, seeking for a breakpoint. If such a breakpoint is found, or the step over condition is fulfilled, then virtual execution pauses and highlights that line. Otherwise the new event, which is either a subsequent instantiation or the end of the current one, is processed. Depending on the actual command condition virtual execution
pauses here or continues in the new context. Instantiation stack and trace view is refreshed whenever virtual execution pauses.

The advantage of an IDE extension for TMP debugging is the high level of integration of the existing framework. The user can use the very same environment as for run time debugging. From implementation point of view we have to implement only the new functionalities, source code visualisation for example can be used “off the shelf”. Better integrating with the IDE also can help in synchronising compiler settings for consistent instrumentation. One of the drawbacks of a plugin solution is its strong dependency on the specific IDE. This can lead to problems when targeting a different platform or different versions of the same IDE, especially if the plugin API is not stable enough and the peculiarities are used. A plugin solution is often limited by the possibilities of the plugin API. Features that could otherwise be simply implemented may become very difficult in such a
constrained environment.

### 5.4.2 Standalone IDE

My second example of a trace based template metaprogram debugger is a standalone application that is very similar to run time IDEs. It operates on C++ projects containing multiple source files, allows editing them and debugging their template metaprograms. Here the target was to create a platform independent environment for TMP debugging while keeping the structure of the application as close to run time IDEs as possible.

Figure 5.3 shows the TMP IDE in a debugging session. We can find the very similar functionalities that we have already seen with the IDE extension: TMP breakpoints, trace event view, instantiation stack, source code view, source tree.

Having a dedicated tool for TMP debugging assumes that we have another environment where the run time behaviour, the standard compilation process is dealt with. It is important to keep these two environments synchronised. In this application, for simplicity, we rely on the user specifying both the compiler and the exact compilation parameters correctly. A more advanced solution could be specifying only a project file of the host IDE, and retrieve the settings from that host project file automatically. Even the source tree could be read from it. Such a close integration is not possible with make based projects. There the source tree and the compilation process is coded implicitly in the makefiles.

Implementing a separate TMP debugger allows it to be platform independent, to show a consistent graphical user interface across different compilation environments. As opposed to an IDE extension the workflow or other functionalities are not limited by an external API. Keeping the project settings consistent with the host environment is important, but difficult. This is the most problematic issue of this approach.

### 5.4.3 Templar

Templar [14, 180] is another standalone application for TMP debugging. In Templar the similarity to a standard IDE was not a goal, which makes it fundamentally different from the IDE version. Templar is basically a TMP trace file analyser. We can explore the sequence of TMP events without explicitly binding to the host project. Though a trace file can contain other kinds of entries, like warnings or errors, Templar ignores everything that is neither a template instantiation nor a memoisation event.

The main screen (Figure 5.4) always shows a specific event of the TMP trace. Details of the event, such as its type (instantiation or memoisation), the name of the template instance, and the source code position, are displayed below the source
CHAPTER 5. TEMPLATE METAPROGRAM DEBUGGING

Figure 5.3: Debugging the classic Factorial example with the standalone TMP debugger.
5.4. APPLICATIONS

Figure 5.4: Debugging the classic **Factorial** example with Templar.

code view. The source code context is shown in the source code view on the left, while on the right we can see the corresponding instantiation graph. Hexagonal nodes represent instantiated templates, while ellipses denote instances accessed through memoisation. As we step forth and back in the event sequence, the colours of the nodes change accordingly. Not yet visited instances are marked light greyish-blue and green colour represents instances that we have already passed. Began, but not yet finished instantiations are coloured dark orange. Below the graph we can see the instantiation stack on the right side.

Events being the central entity rather than the source code, breakpoints can be set on template instances and not at source code locations. In the breakpoint window (Figure 5.5 we can edit a list of regular expressions representing breakpoints. By using regular expressions for matching the name of the instance we have big freedom in specifying breakpoints on templates. We can select specific instances easily, but we can also leave some of the parameters open. Note that this kind of textual match has some limitations, for example a query of `Template<int,.*,int,.*,int>` will select `Template<int,std::map<int,int,less<int>>,long,char,int>` which was probably not the original intent. Using `[^,]*` instead of `.*` would not work either. Apart from these problematic cases using regular expressions is a simple and convenient way of setting
breakpoints.

Buttons \(\leftarrow\) (prev) and \(\rightarrow\) (next) advances to the previous and the next trace event respectively. They are analogous to the \texttt{step} command of run time debuggers. \(\leftarrow\) (rewind) and \(\rightarrow\) (forward) step iteratively until either a breakpoint is hit or one end of the event list is reached. These commands are analogous to \underline{continue} in run time debuggers. Unlike in run time debuggers we can freely jump \underline{back and forth} in the event sequence, going backward is just as simple as going forward.

The \(\bigcirc\) (follow) icon is a toggle. If on, the graph view is automatically repositioned to always show the actual instantiation in the middle.

With the \texttt{Filter} button we can add ignore expressions to exclude trace events from the debugging session. Similarly to breakpoints we can specify them with regular expressions. Trace events matching the regex are virtually deleted from the trace file, as if they did not happen at all. This functionality helps removing distracting instantiations either from the graph view and from the sequence of events we step through. \texttt{Reset} clears all breakpoints and ignore expressions and jumps to the beginning of the trace as if we have just loaded it.

\texttt{Templar} proved to be a handy tool for TMP debugging. Being designed specifically for analysing trace files is a clear advantage over the mixed solutions that try to mimic the behaviour of a run time debugger. The learning curve is not steep, the usage is quite straightforward once the concept is clear.

### 5.5 Related work

The motivation section (5.1) describes why C++ template metaprogramming needs a debugger. Current tools do not fulfil the requirements of a developer who uses C++ templates to create algorithms that run in compilation time. We can make improvements in two basic directions. We can accept template metapro-
gramming as it is and add support to it in the development frameworks, by external tools, or by special purpose programming constructs made from existing language elements. In addition we can work on the language itself, making it directly support the need for programming compile time algorithms. There are results in both directions.

Alexandrescu made compile time assertion available by using partially implemented templates [129]. Some of the work related to template metaprogram instrumentation (see 4.6) are relevant to TMP debugging as well. These are Tracer and archetype of Vandevoorde and Josuttis and the diagnostic section in [47] including mpl::print.

MetaShell [181] from Ábel Sinkovics is an interactive template metaprogramming shell, where compile time expressions are parsed on the fly and the resulting type is output in a simplified form. Custom formatters can be added right through the interpreter by specialising the metashell::format metafunctor.

In the new C++ standards (C++11 [21] and the upcoming C++14 [23]) significant steps are made towards addressing the original need behind template metaprogramming. Compile time assertion became a language element in C++11 [102]. C++11 introduced the constexpr keyword that can mark variables and (member) function to be evaluated in compile time. While in C++11 the use of constexpr was rather limited, a short list summarised what is possible in a constexpr function, most of these limitations have been deleted in C++14. In C++14 the list contains the disallowed elements instead, making everything else allowed.

McNamara [182] and Siek [183] contributed to static interface checking, which seems to become a major new feature of C++ templates, called concepts. The basic idea of constraining template parameters already exists in other languages supporting generic types [104, 105, 106]. With concepts the restrictions are formulated in the form of possible expressions among the template parameters. This is a much more general approach that most other languages have chosen, where much simpler constraints can only be defined parameterwise, a prescribed base type for example. Bigger complexity and the accompanying difficulties in adding it to the language made concepts be left out from C++11. In C++14 a much simpler version is expected to be added, which is called template constraints [133]. With static interface checking the programmer can better express the intentions behind a generic construct. The compiler then is able to report erroneous usage at the point of invocation rather than deep inside the implementation in a conflicting expression. Apart from better diagnostics, this new language feature improves code readability a lot by documenting the interface.

Run time being equivalent to compile time in TMP easily leads to confusions when we want to talk about errors. Running out of (compiler) resources (heap memory, recursion stack) is a typical run time error of a template metaprogram,
arising at compile time. When our metaprogram fails to run because of fundamental compilation errors, we can say our TMP is ill formed. It can happen that a compilation error occurs deeply in a long instantiation chain when many compile time computations have already been performed. Then we have a run time error, our metaprogram was already running after all, due to a compilation error of ill formed code. In his thesis on effective C++ template metaprogramming [16] Ádám Sipos creates a classification of TMP errors to clarify these situations.

Templar (see 5.4.3) by Zoltán Borók-Nagy is getting more and more popular. Sven Mikael Persson generalised the previous implementation by creating a Template Instantiation Observer in clang. He developed a gdb-like command line interface [184] for TMP debugging using this observer of template instantiation events. The corresponding modifications in clang (Templight trace generation [185] and the observer [186]) are expected to become official features of the compiler making the method de facto standard.

### 5.6 Contribution

**Thesis 3** (Methods for template metaprogram debugging). I developed two methods for implementing debuggers for C++ template metaprograms. Both methods are based on a trace file describing the template events. Besides the non-intrusive data extraction method that utilises the instrumentation technique of Thesis 2, I presented one that is based on compiler modification, and has some advantages over the non-intrusive version. I showed three applications that demonstrate how my methods can be used to implement template debugger functionalities. An implementation based on my research is becoming de facto standard for template metaprogram debugging [14, 180, 184, 185, 186].

<table>
<thead>
<tr>
<th>thesis name</th>
<th>relevant publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of unity build</td>
<td>[1] [2]</td>
</tr>
</tbody>
</table>

80
Chapter 6

Template metaprogram profiling

This chapter discusses TMP profiling in details. First, Section 6.1 describes the motivation for a TMP profiler and identifies some typical use cases. Section 6.2 follows by presenting different approaches of implementing a TMP profiler. Important aspects of the different methods are analysed in Section 6.3 showing measurement results. Then Section 6.4 concludes by clearly outlining the pros and cons of each presented approach. Related work on the topic is collected in Section 6.5 and my contribution is summarised in Section 6.6.

6.1 Motivation

According to Knuth [187] less than 4% of a program usually accounts for more than 50% of its run time. Bentley [188] described a case when a 1 KLOC program spent 80% of its run time in a square root routine of 5 lines. These results show the importance of measurement before optimisation. Finding key bottlenecks requires detailed measurements to see where time is actually spent. This is the purpose of a profiler is supporting these measurements. Optimising build time is crucial in large C++ projects (see 2.1 and [189]). Template metaprograms are programs that run in compile time, therefore their performance directly affects build times.

Soon after template metaprogramming gained popularity its significant effect on compile time became a warning both for developers using the new paradigm and for projects to calculate with this risk [189]. TMP has originally not been designed to support TMP [41]. TMP is much more a side effect instead, therefore compilers were not optimised to run complex metaprograms efficiently. This led to functionally equivalent metaprograms implemented in slightly different ways showing different efficiency characteristics [47, 190, 191]. These characteristics also vary from compiler to compiler [47].

When we are talking about compiler efficiency, it is not only about compi-
lation time. Memory consumption of the compiler can become a blocking issue with complex metaprograms [191, 192]. With today’s multicore architectures a parallelised build can easily consume all the RAM of the computer if each parallel compilation process happens to use large amount of memory. If ran out of memory either the compilation fails or the operating system starts to use secondary storage rendering the compilation very slow. According to my personal experience CPU usage can drop down as low as 0.1% from above 95% if memory is served from a slow secondary storage such as a non-solid hard drive. Even if the standalone compilation time of our TMP is satisfying, memory efficiency factors may require its restructuring.

What was discussed so far shows that TMP development needs a good pro-filer at least as much as run time program development for efficient applications. Similarly to run time programs not only speed, but memory efficiency is also an important concern. When optimising our template metaprograms we have to find an optimal algorithm that uses less instantiations and less complex template constructs. These optimisations are presumably worthwhile with all compilers. We also have to take into account the special efficiency characteristics of the used compiler and choose among the possible implementation variations accordingly. Having tools that give instant feedback on our optimisation efforts is essential.

Heisenberg’s uncertainty principle [193] (and Ziv’s uncertainty principle [194], which is Heisenberg’s applied to the software engineering process), that is measuring a system always modifies the behaviour of the system itself, holds for performance measurements as well [195]. Reducing distortion usually requires extra effort in the measurement method. The presented TMP profiling methods differ in the detailedness of the result, the precision/distortion of data, the required work and skills for their application.

### 6.2 Different approaches of implementing a TMP profiler

This section shows four different methods for measuring TMP performance. Each is assigned a two letter abbreviation for short reference later:

- **FC** measure full compilation time
- **WO** instrument warning generating code fragments and measure times out-of-process
- **WI** instrument warning generating code fragments and measure times in-process
- **BI** built-in support for profiling

The in-process/out-of-process distinction is borrowed from the field of COM components [196]. Figure 6.1 shows the workflows of the four TMP profiling ap-
6.2. DIFFERENT APPROACHES OF IMPLEMENTING A TMP PROFILER

6.2.1 Measure full compilation time (FC)

The most common technique is available in all compilation environments: simply measure the full time spent in the whole compilation process. In their book Abrahams and Gurtovoy devote a full section for measurements of this kind [47]. This method will not give us a detailed report, but still it can be used to analyze how our compiler scales when using certain language constructs. Let us suppose we would like to detect how increasing the number of template parameters affects performance. Then we create a generic source file that contains templated constructs having $N$ template parameters, where $N$ is a compile time variable defined when invoking the compiler. By measuring compilation times with increasing $N$ we can create a chart showing the relation between template parameters and compilation time.

The measured times are gross times including not only C++ compilation but also preprocessing, code generation, file I/O, the initialisation and deinitialisation of the compiler backend. These values are far from being precise, but for showing trends of linear or polynomial complexity we do not need high precision. Ease of use and instant availability are the major strengths of the FC method.

6.2.2 Warning instrumentation and out-of-process measurement (WO)

FC can only be used for measuring trends of specific constructs with appropriately prepared source files. When we have a TMP system and would like to pinpoint critical parts regarding performance, FC is not going to fulfil our needs. With TMP instrumentation (Thesis 2) we can artificially make the compiler emit a warning message whenever it instantiates a template. The WO method consists of using TMP instrumentation and recording the actual timestamp every time a warning message appears on the compiler output.

It is recommended to implement timestamping as a simple text filter [197] and add timestamp parsing to the warning parser (see 4.4.6). Then the times can be directly registered to the given template instance in the resulting trace file. When the output of the compiler is redirected to a process, the timestamp decorator in our case, the operating system may change buffering parameters. While in interactive mode a newline character typically makes the output be flushed, this is not the case in non-interactive mode. This difference can ruin profiling data entirely, because a bunch of earlier warnings are registered together with the one that made the buffer full and triggered a flush. To overcome this problem and
Figure 6.1: Workflows of the FC, WO, WI and BI TMP profiling approaches.
eliminate such delays buffering parameters of the pipe between the compiler and the timestamp filter should be changed.

Instrumentation modifies the code, which can distort the measurement. Generating and printing a warning message is an additional delay that is indistinguishable from TMP run time in this method. It is important to keep these distorting factors in mind when using the WO approach. The advantage of WO is its ability to create a detailed performance report and its easy adaptation to a specific compilation environment.

6.2.3 Warning instrumentation and in-process measurement (WI)

Based on the experience with the FC and WO methods, to get a detailed but also precise profiler the modification of the compiler seems unavoidable. The delay of interprocess communication, the generation and output of warning messages are indistinguishable components of the times measured by WO. The WI method filters out these distortion components. WI mixes full built-in support for profiling (BI) with WO. While BI needs at least a rough understanding of the template mechanism of the compiler, WI requires only a simple modification at the warning generation part, which is typically a much more easier, straightforward task. In WI the compiler is modified to print two timestamps for each warning: one at the beginning of warning generation and one at its end. The warning parser extracts these timestamps from the compilation output and registers them to the given template instantiations just like the single timestamps in WI. The difference of the two timestamps gives the time spent in warning generation, something that we do not want to include in the measure. Further analysis of the trace file can use these two timestamps to display a more correct profiling information by subtracting warning generation times from total measured template processing times.

This approach still has a distortion factor because of instrumentation, but at least warning generation does not increase distortion. To get the best results the implementation should flush output buffers before determining the end timestamp to make sure that every task related to the warning output does happen in the measured period. High resolution timestamps are vital for this method to work. Not only the resolution matters, but also obtaining the value should not take too long to avoid further distortion. Fortunately on most platforms there is a preferred fast way of retrieving a high resolution timestamp [198].

6.2.4 Built-in support for profiling (BI)

Instrumentation based methods work with modified source. Performing this source code modification at each occasion of profiling is inconvenient. Furthermore, in-
CHAPTER 6. TEMPLATE METAPROGRAM PROFILING

Instrumentation causes hardly predictable distortion effects on the measurement. A built-in solution does not require instrumentation and the distortion is small. Similarly to the native TMP debugger approach (see 5.3.2), template instantiation code should be identified in the compiler and modified to collect performance data. The hook points listed in Table 5.5 are appropriate for profiling too.

The recommended implementation is to simply register each template related event along with a timestamp.

For precise results a high resolution and fast timestamp generator is important in this method as well. Moreover, the administration of the timestamp and/or memory usage should also be quick. Our solution, for example, uses a pre-located fixed size array, where the insertion of new data has $O(1)$ run time cost. Initialisation of the array happens before any part of the compilation process start, and the collected profile data is written to file only when the compilation has already finished. Using a fixed size array has user side consequences (the user has to adjust the array size through a command line parameter if needed). An alternative approach could be to exclude the time of administration, including the allocation of new space, from the measurement. This latter technique is very similar to excluding warning generation time in the WI method (see 6.2.3).

BI is not an option if either the source code of the compiler or the competence for adding profiler functionality to it is not available.

6.3 Analysis

In this section various measurements are presented that highlight different characteristics of the described TMP profiling methods. The order of the analysed approaches intentionally differs from the previous less intrusive to most intrusive FC, WO, WI, BI order. Most measurements are based on the BI method, therefore the analysis of BI precedes the discussion of the measurements based on BI.

6.3.1 Example codes used in measurements

Two source codes were used in the measurements. One of them is an adapted version of a special purpose code from Abrahams and Gurtovoy [47], while the other one is an open source library. Both source codes contain template constructs that are standard techniques in TMP.

In [47] efficiency of different language constructs are measured using the FC technique. These measurements are often comparative. First a baseline code is measured to create a reference, then the code is modified to contain the measured language construct and measured against the reference. We will refer to the modified version as measure as it is the actual measurement.
One such measurement compares the efficiency of using inheritance or plain reference to create recursion in a simple template metaprogram. Listing 6.1 contains the test code, where BASELINE enables the baseline version. By invoking test with different arguments the same recursion is executed ten times in the TEN_TIMES version to reduce the deviation of the measurement. The deviation, also known as measurement bias [199], has different reasons [200] from environmental aspects (OS task scheduling, memory fragmentation) to the observer effect [201] (the presence of the measurer changes the behaviour of the system). Measuring the full compilation time for increasing Ns one by one both for the baseline and for the measure version results in a chart showing the characteristics of the two versions. As it is shown in [47] the inheritance version has cubic complexity while the reference version seems to be linear.
Bayes++ [202] is an open source C++ library for Bayesian filtering. It employs Boost.uBLAS [203], a basic linear algebra library that heavily utilises template metaprogramming techniques.

6.3.2 Measuring preprocessing overhead of FC

In FC not only the actual TMP processing is measured, but all the steps the compiler invocation consists of from loading the process, executing the preprocessor, parsing C++ code, generating code to linking then writing the generated binary to disk. References to external files in `#include` directives are resolved in the preprocessing phase. `#included` files are looked up in the specified include directories, meaning that file system efficiency can have big impact on the measured times. Figure 6.2 show our results for Bayes++ when measuring full compilation times and when the preprocessing phase is executed separately. On the chart the results of different units are sorted by their full compilation times. This technique is used throughout the chapter to ease the comprehension of the charts.

We can see that preprocessing time is significant and varies. A possible improvement of the FC method could be to perform preprocessing separately and measure the compilation of the already preprocessed source. The average standalone preprocessing time is 1.64, while the average difference between the compilation times of original and preprocessed sources is 0.73. Performing preprocessing separately therefore takes 0.91 seconds more time than in one step in our example. As we saw in 2.2 preprocessed sources tend to be really huge compared to the original source code. When preprocessing is done together with compilation this huge data is transferred from the preprocessor to the compiler in memory. In case of

Figure 6.2: Duration of full compilation versus preprocessing separately for the units of Bayes++. The results of different units are sorted by their full compilation times.
separate preprocessing that huge data is written to and loaded from disk, causing I/O costs. I/O costs can contain HW, OS and SW (serialisation and deserialisation) components. We think the almost one second difference between one step compilation and separate preprocessing originates partly from this I/O overhead. The compiler does not know that the input code is already preprocessed, therefore we should calculate with the cost of preprocessing the already preprocessed code. We think this is the other part of the difference on top of I/O costs.

6.3.3 Using BI for the inheritance/reference comparison

Our target was to use the BI method to reproduce the comparison of template nesting efficiency between inheritance and reference TMP techniques.

We worked with gcc but our implementation can be easily transferred to other compilation environments. The hook points are general enough to be found in other compilers. The idea is the following: we clearly identify the beginning and end of template instantiations in the code of the compiler, and add timestamp recording to these points. We should do this in a way that minimises its influence on compilation time.

Gcc has a(n adjustable) limit for the maximal template instantiation depth because of efficiency considerations and to avoid endless loops. The counter of the actual instantiation depth is increased whenever an instantiation begins, and is decremented when ends. These are exactly the points in the code that we need for profiling. For different technical reasons certain instantiations of a template can be deferred. The gcc source calls them pending templates. When processing pending templates, it is important to simulate the original context by restoring the same instantiation depth that was at the moment of deferral. We have to take this into consideration when maintaining our own profiling data structure. Also, we have to prepare for multiple instantiations of the same instance, since the instantiation itself is not necessarily an atomic process, it is sometimes done in parts. Table 6.1 shows the whole process with our hooks added.

By now we see how timestamps are determined. Let us see how template instances are identified. Internally template instances are referenced by pointers. At the point of instantiation we could retrieve the name, but that would take time and therefore would cause overhead that we do not want. A simple answer could be to retrieve the names at step 7. At that point, however, it is not guaranteed that all data structures are still in memory, calling the name retrieval to the saved pointers would easily result in a crash. We exploit the deterministic property of the compilation process and execute the same compilation two times. In the first one we only gather timestamps, adding them to a global array without any memory handling overhead. In the second pass we do not have to be careful with the overhead, because we already have timing data. Therefore we can print the
**CHAPTER 6. TEMPLATE METAPROGRAM PROFILING**

<table>
<thead>
<tr>
<th>nr.</th>
<th>event</th>
<th>function</th>
<th>what we do</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>file begin</td>
<td>cxx_init</td>
<td>init</td>
</tr>
<tr>
<td>2</td>
<td>*template begin</td>
<td>push_tinst_level</td>
<td>store timestamp</td>
</tr>
<tr>
<td>3</td>
<td>*template end</td>
<td>pop_tinst_level</td>
<td>record instance</td>
</tr>
<tr>
<td>4</td>
<td>file end</td>
<td>instantiate_pending_templates</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*template begin</td>
<td>push_tinst_level</td>
<td>store timestamp</td>
</tr>
<tr>
<td>6</td>
<td>*template end</td>
<td>pop_tinst_level</td>
<td>record instance</td>
</tr>
<tr>
<td>7</td>
<td>file done</td>
<td>end of instantiate_pending_templates</td>
<td>print results</td>
</tr>
</tbody>
</table>

Table 6.1: Hook points when applying the BI method. Lines marked with * can be executed multiple times.

1 `[0];1274743019.026875;1274743019.026875;0.000000
2 `[1];1274743019.025875;1274743019.026875;0.001000
3 `[2];1274743019.025875;1274743019.027875;0.002000
4 `[3];1274743019.025875;1274743019.027875;0.002000
5 `[4];1274743019.024875;1274743019.027875;0.003000
6 ...  

Listing 6.2: Output of run #1 (no print version) when applying the BI method.

names of the instances right at the end of their instrumentation, at the point where they are in memory for sure. As a result we will have two outputs, one with the numbers, and one with the names. We can get the final table simply by putting these two together. For simplicity we decided between the two runs with the lowest bit of the instantiation depth limit, which was already a parameter. This way we did not have to introduce another command line parameter. Listing 6.2 and 6.3 show the output of run #1 and #2 respectively, TEN_TIMES was not defined. In output #2 we have mapping between the ids and the names first, only then follow the times.

We measured our example with the direct compiler support version. We could successfully perform the same test with 1x2 compilation that originally needed one for each N. Figure 6.3 clearly shows both the linear and polynomial trends. Note that the print and no print versions are literally the same. There is a minimal difference, but it is negligible in the scale of the plot in this special example. In real world measures, however, even this minimal difference could count, so it is
6.3. ANALYSIS

Listing 6.3: Output of run #2 (print version) when applying the BI method.

Figure 6.3: Measurement results of the BI method. The no print version refers to the more precise run #1, while print denotes run #2, where template instance names are looked up and printed.

advisable to measure in two passes.

In the rest of this chapter the BI method is used to measure different properties of other profiling approaches. The \(O(1)\) operations that were added to the code for collecting profiling data are negligible compared to the original steps performed by the compiler at the hook points. This allows us to assume that the BI method produces precise results.

6.3.4 Measuring instrumentation overhead

One of the drawbacks of using WO or WI for profiling is their overhead caused by instrumentation. To investigate the magnitude of this overhead we instrumented
the same example and measured the modified code. The result are shown in Figure 6.4. The *print* and *no print* versions are again almost the same, but instru-

![Figure 6.4: Instrumentation overhead.](image)

mentation simply doubles the compilation time, which means +100% overhead. Although this overhead should be taken into account in real projects, we can see that the asymptotic characteristics of the curves are the same in the instrumented and non-instrumented versions.

Let us examine the beginning and the end of the curves in Figure 6.5. We can

![Figure 6.5: Instrumentation overhead for low and high values.](image)

discover a sudden jump in the high values of the *print* version, which is presumably caused by some input/output buffer flush.

Note that so far we only added the extra code fragments, but the generation of warnings was not enabled in the compiler. Figure 6.6 shows what happens

![Figure 6.6: Instrumentation overhead.](image)
if warning generation is on (**-Wwrite-strings**) and therefore extra lines appear on the compilation output. The figure contains only *no print baseline* variants

![Figure 6.6: Instrumentation overhead when warnings are enabled. We can observe the effects of adding instrumentation first, then enabling warning generation, and finally adding timestamp generation to the warnings.](image)

and show how the overhead builds up as we enable more and more components of the instrumentation based profiling methods: instrumentation, warning generation and timestamps in warnings. As we can see the overhead is really huge (at $N = 1800$ it is +642%) and we have lost the originally linear characteristic of the curve. The measurements based on warnings misleadingly show a nonlinear behaviour. As seen in the figure, adding timestamp output at warning generation in itself does not noticeably change the curve. This means that the warning based approaches would be practically useful if instrumentation and warning generation did not produce significant overhead.

As described at the WI method we may expect more precise result if warning output is measured separately and subtracted from the output timestamps in the post processing stage. Unfortunately this does not seem to help, since Figure 6.6 already contains the result of this enhanced method of WI. Figure 6.7 shows these separately measured warning output times, i.e. how long the calls to warning output took for different $N$s. Comparing it with Figure 6.6 we can assert that this estimation is not able to capture the real difference that comes from enabling the warnings.

The last few figures dealt with the *baseline* version. Figure 6.8 shows the difference between raw and corrected times in the *measure* version. Raw times are simply the difference between the timestamps at the warnings that identify template beginning and end. In the corrected values the time spent in warning generation is separately measured and subtracted from the raw time. While the
raw curve seems to be linear, the corrected one is more similar to the cubic curve we should get.

The code examined above has a speciality of deep template recursions. This might be the root cause of the big instrumentation overhead we experienced. Now let us see the overhead in case of Bayes++. Figure 6.9 shows summarised times of normal compilation, i.e. without any instrumentation, then with instrumentation having warnings turned on, and when warning generation times are subtracted. We will refer to this latter as corrected. While these are summarised times, Figure 6.10 shows the instrumented (total) and corrected times for individual template instances. The 200 most time consuming instances are shown, first ordered by total, then by corrected times. In the figures the same template instance may occur several times if it was instantiated in multiple compilation units. We can observe the effects of using a low resolution timer producing a staircase function, only
Figure 6.9: Normal, instrumented and corrected times of Bayes++ compilation using the WI method, summarised by compilation units. D and R stand for debug and release versions of the same unit respectively.

Figure 6.10: Total times and corrected times of the 200 most time consuming template instances of Bayes++ compilation using the WI method. The left figure is ordered by total times while the right is by corrected times.

multiples of the timer resolution are occupied by values.

6.3.5 Differences between debug and release builds

Figure 6.11 shows how much time is spent in processing templates when compiling the different units of Bayes++ both in debug and release modes. The measurement was made by applying the BI method with a modified g++ compiler. Though the full build times are similar in debug and release, surprisingly the ratio of the time spent in templates is much lower in release. A possible explanation could be
that generating debug information for templates takes significant time in debug mode, which is compensated by the extra optimisation phase of release mode thus producing similar full times.

Another interesting question is the multiplicity of template instances among the modules. The same template instance can be compiled in multiple modules, the histogram of this multiplicity for Bayes++ is shown in Figure 6.12. The curve rapidly increases at the end, which means there are many instances which present in all modules. Had these instances generated only once, in a precompiled header for example, we probably would get better build times.


6.4 Conclusion

In this section the numerous measurements we have seen are evaluated. Table 6.2 compares the implementation requirements and the precision of the four profiling methods. As we go from FC to BI the implementation requirements increase, but in exchange we get more and more precision.

According to Figure 6.11 apart from template processing significant time is spent in other compilation tasks, especially in release builds. These other tasks are preprocessing (as shown in 6.2), processing of non-template parts of the code, code generation, linking. Consequently FC is not appropriate for precise template profiling. On the other hand asymptotic time characteristics of specific template constructs can be determined even with the FC method, FC results matched those of the more precise BI method in our measurements (Figure 6.3).

The WO and WI methods try to mix FC and BI: get more detailed results, but without fully understand the internal mechanisms of the compiler. According to Figure 6.4 only adding the warning generator code fragments without turning the warnings on already produces significant overhead. Figures 6.6 and 6.7 show that though in WI the warning output time is measured, it can capture only a small fraction of the big time difference caused by enabling the corresponding warning. Even though the baseline/measure example may be a bit special with its long instantiation stacks, it clearly points out the inaccuracy of the approach. Subtracting warning generation times can save the curve shape of the measure version (Figure 6.8) and can reconstruct the original build time of Bayes++ (Figure 6.9). These cases and Figure 6.10 demonstrate the advantage of WI over WO. In Fig-

<table>
<thead>
<tr>
<th></th>
<th>FC</th>
<th>WO</th>
<th>WI</th>
<th>BI</th>
</tr>
</thead>
<tbody>
<tr>
<td>implementation difficulties</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>output parsing</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instrumentation</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compiler modification</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>deep understanding of compiler</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>precision</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>indistinguishable template instances</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-compilation times included</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instrumentation overhead</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>pipe overhead</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>warning generation overhead</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Implementation requirements and precision of different TMP profiling methods.
Figure 6.10 we can also witness the importance of using a high resolution timer. The figure contains the 200 most time consuming instances, and at the lower end the measured times nearly match the coarse resolution of the used timer.

The BI method has the lowest overhead, where template instantiations are registered directly as part of the compiler functionality. In BI the overhead consists of timestamp retrieval, template event registration and optionally printing the measured values. For timestamp retrieval fast methods exist [198], event registration is a simple assignment and index increment if preallocated vector is used. Printing the measured values did not cause noticeable overhead either in our measurements (Figures 6.3, 6.4 and 6.5). The only exception is a sudden jump in the print version in 6.5 presumably due to output buffer flush.

Figure 6.11 and 6.12 teach us not to underestimate the effect of changing build configuration to template processing time and the number of instantiations. Multiplicity distribution has nothing to do with measured times, however, it can be effectively used for build time optimisation by putting the determined widely used template instances into a precompiled header.

Note that the measurements have been performed years ago with old compiler versions. They were good for pointing out numerous interesting properties of template profiling that we should be aware of when choosing among the methods. The quantities and ratios can vary highly from compiler to compiler, therefore it is always advisable to do some fresh measurements with the actually used compiler environment. The presented analysis may be improved by repeating it with a wider set of code bases, build configurations and compiler environments.

The Templight trace generation in clang (see 5.5) is becoming more and more popular among TMP developers not only for debugging, but also for profiling [180, 184] (see 6.5), which proves the industrial applicability of the BI method.

6.5 Related work

Relatively few works exist that address TMP profiling. Though there is a demand from developers to have powerful profilers for template metaprograms, compiler vendors do not give real solutions to the obvious needs (feature request closed as “Won’t Fix”: [204]).

Abrahams and Gurtovoy devoted a whole chapter for compile-time performance in their book [47], where they used the FC method to analyse various aspects of template processing in different compilers (see also 4.6, 6.2.1 and 6.3).

Steven Watanabe’s template profiler (see 4.6) is very similar to WO. Instead of measuring times it counts template instantiations.

Zoltán Borók-Nagy, the author of Templar (see 5.4.3) implemented Profile-DataViewer, which is a viewer for time information stored in a Templight trace file
6.5. RELATED WORK


(see 4.4). Listing 6.4 shows how timestamp appears in a trace entry. For a complete trace file with timestamps refer to Listing C.1. Timestamps in TemplateBegin and TemplateEnd entries are used to calculate the time spent in the given template instance. Either the WI, WO or BI methods can be used for generating such trace files that contain the necessary timestamp entries. Figure 6.13 shows the two tabs of the application that display template instance times structured by dependency or by scope respectively.

![Figure 6.13](image)

Figure 6.13: The two views of ProfileDataViewer. Dependencies show template instance times according to their caller/callee relationship, while Namespaces follows scopes (nesting namespaces and classes).
### 6.6 Contribution

**Thesis 4** (Methods for template metaprogram profiling). I identified one existing and three new methods for implementing profilers for C++ template metaprograms with different strengths. Supported by detailed measurements I pointed out important aspects of applying these methods in practice. An implementation based on my research is becoming de facto standard for template metaprogram profiling [14, 184, 185, 186].

<table>
<thead>
<tr>
<th>thesis name</th>
<th>relevant publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of unity build</td>
<td>[1] [2]</td>
</tr>
<tr>
<td><strong>Methods for template metaprogram profiling</strong></td>
<td>[9] [10] [11] [12]</td>
</tr>
</tbody>
</table>
Chapter 7

Type-preserving heap profiler for C++

This thesis chapter focuses on how to preserve type information and make use of it in a C++ heap profiler framework. First I define the requirements that the profiler framework should fulfil in Section 7.2. Then an overview of the profiler framework and its main data structures are discussed in Section 7.3. Section 7.4 walks through the subtleties of the implementation. In Section 7.5 we get familiar with the usage patterns of the toolset. Section 7.6 describes what kinds of analysis and visualisation is possible once we have the profiler output. When discussing the visualiser tool, efficient algorithms are presented that allow quick visualisation of big data sets. Section 7.7 presents a case study showing a few typical use cases while Section 7.8 presents similar works focusing on the differences. Finally my contribution is summarised in Section 7.9.

7.1 Motivation

Most object-oriented programs heavily use heap operations [205]. Objects with dynamic lifetime are created on the heap. There are numerous design patterns [128] which require one or more of their components to be located on the heap. Heap operations are inherently slow compared to other in-memory activities [205, 206]. They usually go through several layers and manipulate a complex data structure. Sometimes an allocation has to perform garbage collection and/or even file operations (swapping). Thread-safety is another bottleneck regarding effective heap usage [207]. Because of their high performance footprint, the optimal usage of heap operations is crucial when optimising applications for speed.

According to [187, 188] in most cases only a relatively small amount of code is responsible for the majority of application running time. Therefore it is very im-
important to know what parts of the code these are. Profilers can help us identifying them.

There are many good heap profilers for different platforms. They typically hook heap allocation routines (malloc and free on many platforms) and monitor memory related activities. With these tools we can freely explore what call stacks belong to the allocations. One property, however, is lost in almost all C++ memory profiler tools we know: what type of object the allocation has created. Type related information about heap usage can be very useful in modern object-oriented languages. As types, classes are the main language construct, therefore type information can reveal hot-spots, critical bottlenecks on a higher abstraction level.

For those languages, where a reflection model is built into the language, retrieving heap related type information is easy. This is the case in Objective C [208] and Java [209] for example. The C++ programming language is infamous about its very poor reflection capabilities [210]. C++ does not give much aid for preserving type information for heap operations.

Tools operating on binaries have minimal chance to recover such data in C++ since the very same code is generated for different kinds of objects with same size and they are not differentiable afterwards. Also, in C++ only minimal run time information are kept about the individual objects. A solution, therefore, probably requires a separate build configuration where these information live longer.

Once we retrieved profiling related data we want to use them to better understand program behaviour. Contrary to the low level infrastructure types, like strings, data structures, thread synchronisation primitives, many classes can be directly connected to certain program features. Therefore type information can naturally group memory allocations to features or program components. On resource-limited platforms it is important to know the memory footprints of each individual feature when composing a new product. In some cases features can be easily measured using only call stack information. Large systems, however, often contain complex cooperative components where objects have shared ownership. In those situations we cannot select the exact code area which is responsible for handling these objects.

In this chapter I present a heap profiler framework for C++ that is capable of preserving the type of allocated objects. The framework is based on macros, operator overloading, and instrument functions. To apply it on a project, some constructs in the source code have to be modified, but fortunately this transformation is straightforward and seldom requires significant time. The data retrieved by using the profiler can be analysed in an interactive visualiser tool, where we can focus on arbitrary aspects with user defined filter graphs and special views.
7.2 Requirements

The C++ language is well known of its limited (almost nonexistent) introspection features [210] compared to Java or C#, where the run time system supports reflection. Therefore among the numerous C++ heap profilers we can hardly find any that provides sufficiently precise type information. My target was implementing a type preserving heap profiler for C++.

In this section I enumerate the main requirements against the desired tool. An ideal type preserving heap profiler has the following essential features:

- provide a full overview of the allocations and deallocations with call stacks and types
- preserve exact type information, including template constructs like `std::vector<std::string>` or even more complex ones
- have acceptable syntactic overhead
- work on real, industrial sized C++ systems
- have acceptable overhead both in the means of memory and speed
- support multithreading
- no influence on performance if the profiler is not enabled (zero-overhead rule [211])
- be as platform/compiler independent as possible
- ease of use

7.3 System design

In this section I present an overview of the key mechanisms behind my type-preserving heap profiler. Here the focus is on what the profiler does, and the connection between its major components, while Section 7.4 describes the details of how the different functionalities are achieved.

7.3.1 System structure

Figure 7.1 shows the main actors of my profiler framework. Code represents the client code that we are profiling. This client code performs heap operations with calls to (de)allocation primitives, like `new` and `delete` (or their array forms, `new[]` and `delete[]`) or `malloc` and `free` (see also `calloc` and `realloc` [212]). These allocation primitives end up in an allocator that actually deals with heap memory. Code and Allocator are part of the client system, they connect to the two major component of the profiler framework: Memory logger and Thread monitor.

Memory logger keeps track of the allocated memory blocks in allocation entries. Allocation entries of not yet freed blocks are called active. Memory logger has to be
hooked [213] into Allocator to register (de)allocations immediately. To have call
stacks in the allocation entries, the client code should be compiled with instru-
mntation turned on in the compiler. Instrumentation adds calls to user specified
enter and exit functions at the beginning and end of each function call respectively.
Thread monitor continuously monitors the actual call stack of each running thread
through these instrumented calls. Whenever memory logger registers an alloca-
tion or deallocation, Thread monitor saves a snapshot of the actual call stack into
its Calling Context Forest (CCF) data structure, and a reference to the new (or
already existing) CCF entry is stored in the allocation entry.

7.3.2 Data structures

One of the common use cases of a memory profiler is finding the root cause of high
memory consumption. In such cases there can be only little memory available for
the profiler framework itself. In those cases low memory footprint of the framework
is very important. In other cases we may have plenty of free memory and we would
like to have very detailed information about the heap behaviour of our application.
To serve both use cases I introduced optional fields in the allocation entries. The
user can control memory overhead by selectively enabling the collection of the
following information: timestamps at allocation and free, call stacks at allocation
and free, object type of allocation. Note that if both call stacks at allocation and
at free are disabled, then Thread monitor is not necessary and instrumentation
can be disabled.
7.3. SYSTEM DESIGN

Memory logger and Thread monitor use the following data structures:

\[
\mathcal{E}_a := T_{\text{alloc type}} \times N_{\text{size}} \times N_{\text{count}} \times N_{\text{object type}} \times \mathbb{P}_{\text{alloc time}} \times \mathbb{P}_{N_{\text{alloc stack}}}
\]

\[
\mathcal{E}_f := P_{\text{address}} \times \mathcal{E}_a \times N_{\text{free time}} \times \mathbb{P}_{N_{\text{free stack}}}
\]

\[
\mathcal{A} := P_{\text{address}} \rightarrow \mathcal{E}_a
\]

\[
\mathcal{F} := [0..B-1] \rightarrow \mathcal{E}_f
\]

\[
\mathcal{N} := P_{\text{address}} \times \mathbb{P}_{N_{\text{first child}}} \times \mathbb{P}_{N_{\text{next sibling}}}
\]

\[
C := \text{N}_{\text{thread}} \rightarrow \mathbb{P}_{N_{\text{CCT root}}}
\]

where \(\text{opt}\) denotes optional fields, and \(\mathbb{P}_X\) is a pointer to \(X\). \(\mathbb{P}\) is a pointer to an object on the heap that was allocated by the client code. Timestamps, size, count and thread id are all represented by natural numbers, these are some kinds of integer types in an actual implementation depending on the target compilation environment. \(T\) is the set of allocation types (see 7.5), while \(B\) is the size of the buffer storing freed allocation entries.

Memory logger uses the \(\mathcal{A}\) and \(\mathcal{F}\) data structures to store active and freed allocation entries respectively. \(\mathcal{A}\) is an associative container (preferably a hash table) allowing fast lookup both for addition and removal. In multithreaded environments concurrent access of \(\mathcal{A}\) should be taken into account. \(\mathcal{A}\) contains allocation entries for all heap memory actually being used by the application. This is ideal for creating snapshots at arbitrary time, not only at program end. Freed entries are simply added to a vector, and flushed to disk when the vector is full (reaches its full capacity of \(B\) elements), producing a sequence of dumps describing the past.

By using the instrumentation technique we can have an always up to date call stack for each thread (see also 7.4.4). Let us see how these call stacks are associated with the allocation entries. Copying full call stack information into the entry would consume too much memory. Once entered into a function, all allocations from within that call will have the same call stack prefix. We can optimise our data structures to exploit this property. Storing an evergrowing calling context forest (calling context trees (CCT) [214] of each individual thread) instead of independent arrays is a good trade-off between speed and storage. CCT allows storing common prefixes of different call stacks only once. Each node of the tree represents a function invocation on a specific call path starting from the thread’s
main entry. Multiple tree entries can refer to the same function if that function has been called on different paths. The path to the thread root clearly identifies the call stack, therefore an allocation entry can identify its call stack by storing a reference to the corresponding tree node. This data structure is a trie [215, 216] for storing call stacks. A leftmost-child-right-sibling representation [217] can keep memory footprint low, while insertion remains fast. Thread monitor stores the call stacks of (de)allocations in a calling context tree for each running thread in C, which forms the calling context forest (CCF).

To achieve better performance the CCF is not maintained continuously, instead it is extended only when an allocation entry references its node. When a new allocation entry is registered, the function entries of the actual call stack array are looked up in the CCF and the possibly missing nodes are created. For simplicity unused nodes are not removed from the CCF. This simplification can lead to memory consumption problems if there are too many different call stacks. In our experiments with a code base of some millions LOC we did not encounter such memory consumption problems even when running the instrumented application for 10+ hours. If keeping all entries turned out to consume too much memory, a garbage collection mechanism could be easily implemented that would remove unused nodes. Then a reference count would be necessary in the nodes to indicate that the node is used by the allocation entries.

7.4 Implementation details

7.4.1 Capturing memory operations

Many C++ heap profilers work on binary code using techniques like dll injection or preloading (see 4.1). A tool that operates on compiled binaries would be very useful, but unfortunately the C++ binaries do not preserve type information. Though we can identify calls to malloc and free, the original types are lost. Attempts to recover type information from memory layout is hopeless. Having two structures with identical member layouts, the generated code for object allocation of one or the other will be the same. Debug information, if exists, could help to identify the type, but that would require big efforts for each supported compiler (more precisely each supported debug information format).

Due to the issues listed above it is more promising to address the problem at the time of compilation when type information is still available. Our main targets are the typed new and delete expressions and the typeless malloc calls with their fellow operations: free, realloc, calloc.

In C++ the new expression is the main language feature for creating objects on the heap. The new expression is not equivalent to the new operator. The new
expression is responsible for allocating space for objects on the heap by calling one of the overloaded new operators and for calling the constructor to ensure proper initialisation of the raw memory area. It is also responsible for catching possible exceptions caused by the constructor and clean up the already allocated memory in such cases to avoid memory leaks. The new expression therefore has exact type information, while new operators are typeless.

Though new operators will not help us identifying the types, they are perfect hook points of memory operations. Let us start with redefining operator new. The signatures are the following:

```cpp
void* operator new (size_t);
void* operator new[] (size_t);
void operator delete (void*);
void operator delete[] (void*);
```

For simplicity we omit the discussion of other overloads like the noexcept versions, they can be handled similarly. We left out the placement versions as they have nothing to do with actual (de)allocation.

Apart from calling the actual allocation routines (the standard malloc, or any already existing custom implementation), we decorate these functions with an extra administration step. With this added administration step we can keep track of the address and size of each allocation and its corresponding deallocation (see 7.3.2).

It is important not to use the redefined allocation routines from within themselves. This can be accomplished by using custom allocators or static memory areas in the framework code. Custom allocators should use the low level allocation primitives not garnished with the administration.

Caveat: Heap allocations can happen during static initialisation:

```cpp
A* globalVarA = new A;
```

which means our administration should not rely on static initialisation order. An on demand initialisation technique is recommended.

### 7.4.2 Retrieving type information

Although we are able to monitor allocations and deallocations through the new and delete operators, their types are still missing. The expression new T has a static type of T*, but we cannot reach this type information inside the new operator, which only gets a size parameter and returns the allocated area as a typeless void* pointer. Note that the allocation may pass arguments to one of the constructors, and also the type name can contain commas: new std::pair<int,int>(3,4).
CHAPTER 7. TYPE-PRESERVING HEAP PROFILER FOR C++

1 template<class T>
2 struct TYPE_IDENTIFIER {
3     static char mDummy;
4 };  
5 struct NewTrick {
6     template<class T>
7         T* operator* (T* Ptr) {
8             RegisterAllocation(Ptr, &TYPE_IDENTIFIER<T>::mDummy);
9             return Ptr;
10         }
11     }

Listing 7.1: Implementation of the NewTrick class.

This is important if we are considering using macros, which is a usual technique to minimise syntactic overhead. To demonstrate the difficulties of seizing type information, consider the following attempts:

- \texttt{New((std::pair<int,int>),(3,4))}
  where \texttt{New} is a macro with two parameters
- \texttt{New((std::pair<int,int>)(3,4))}
  where \texttt{New} is a macro with a sequence [47] parameter
- \texttt{New(new std::pair<int,int>(3,4))}
  where \texttt{New} is a template function
- \texttt{new std::pair<int,int>(3,4)}
  where \texttt{new} is a macro, substituting to \texttt{New(new}, yielding the previous line

The last one already has minimal syntactic overhead, though in a complex expression the extra parenthesis can be very confusing. We should eliminate that. Fortunately C++ has the possibility to call a function without writing any parenthesis with its infix operator syntax. Operators are ranked by well defined precedence categories and can be templates as well.

Our solution is \texttt{new std::pair<int,int>(3,4)}, with zero syntactic overhead, where the \texttt{new} macro is defined as follows:

\texttt{#define new NewTrick() * new}

Here \texttt{NewTrick} is a helper type, with an overloaded multiplication operator for each \texttt{T}, see 7.1 for its implementation. The key feature of this construction is the \texttt{mDummy} static member of the \texttt{TYPE_IDENTIFIER<T>} helper template. In \texttt{NewTrick()} * \texttt{new} we call the default constructor of \texttt{NewTrick} to create a temporary object on which we can perform our overloaded \texttt{operator*}. This template \texttt{operator*} takes a pointer of arbitrary type as right hand side operand. In our case this operand
7.4. IMPLEMENTATION DETAILS

```
1 public: static char TYPE_IDENTIFIER<unsigned short>::mDummy 010f29d4
2 public: static char TYPE_IDENTIFIER<unsigned char>::mDummy 010f29d5
3 ...
4 public: static char TYPE_IDENTIFIER<class std::set<int,struct
    std::less<int>,class std::allocator<int> > >::mDummy 01118dec
5 ...
6 public: static char TYPE_IDENTIFIER<struct std::pair<unsigned
    long,bool> >::mDummy
```

Listing 7.2: Type indentifiers in the map file.

will be provided by the `new` expression. Inside the operator the second parameter
of the call to `RegisterAllocation` instantiates the `TYPE_IDENTIFIER` template
class for each type which happens to appear on the right hand side of the `new`
keyword.

For each such type there will be a unique global variable for type identification.
It is important to understand that these variables are not holding any specific
value for identifying the corresponding types like `std::type_info` objects do in
RTTI [210]. It is the existence of the variables with their separate addresses which
holds the information we are looking for. These addresses with the appropriate
type names are listed in the map file generated by the linker. The map file contains
the addresses of all entities of the binary, including these `mDummy` variables. Map
file generation is available on all platforms, although the format of the generated
file may vary. Having the map file at hand, we can associate type names to the
type identifier addresses. Listing 7.2 shows fragments of a demangled [218] map
file.

This way we do not have to generate a string from a type, which would be
a difficult task, if not impossible without syntactic overhead. Our technique also
has a positive impact on the memory overhead of the profiler framework, as the
growth of the executable size and static memory footprint is minimal. In the
optimised version of a concrete mid-sized real application the map file had 3675
`TYPE_IDENTIFIER` entries. This equals to the number of types involved in `new`
expressions. With our method the memory footprint is therefore 3675 bytes (1
byte per each `mDummy` instance). If the mangled or demangled names had been
stored in memory, the memory footprint would have been 191412 or 243222 bytes
respectively.

We chose `operator*` because that was one of the suitable binary operators
with strongest precedence. Thus we avoided unwanted side effects if someone
mixed allocation and pointer arithmetics:

```cpp
ObjWithRefCount* obj = (ObjWithRefCount*)(sizeof(RefCount) + 
    new char[sizeof(RefCount) + sizeof(Obj)]);
```

If we had chosen `operator=` for example, then in this case the preprocessed code would have been:

```cpp
ObjWithRefCount* obj = (ObjWithRefCount*)(sizeof(RefCount) + 
    NewTrick() = new char[sizeof(RefCount) + sizeof(Obj)]);
```

and the compiler would have complaint about missing `operator+` overload between `size_t` and `NewTrick`.

In certain cases a two phase approach is preferable, where the lowest level memory manager routines (called from a redefined `malloc` and/or `operator new`) call `RegisterAllocation` without type information and `NewTrick::operator*` calls a `SetAllocationType` type modifier on the already registered entry afterwards. This implementation has the advantage that if type information is not needed, the whole `NewTrick` trick can be disabled, still keeping track of the allocations via a typeless mechanism.

### 7.4.3 Pitfalls

**Array size** In case of `new[]` the item count is sometimes stored at the beginning of the allocated area. More precisely it is typically done so when the type has a non-trivial destructor to be called for each allocated element at deallocation. The following C++ code:

```cpp
struct A {
    -A() {}
    int member;
} *fiveAs = new A[5];
```

is compiled as:

```cpp
size_t* allocated = (size_t)::operator new(sizeof(size_t) + 
    5 * sizeof(A));
*allocatedArea = 5; // store size
A* fiveAs = (A*)(allocated + 1);
```

Having the `new` macro defined as described, we get this:
7.4. IMPLEMENTATION DETAILS

allocated

\[ \begin{array}{ccccc} 5 & A & A & A & A \\ \end{array} \]

\text{fiveAs}

Figure 7.2: Example of the typical memory layout of an array with non-trivial destructor.

```cpp
size_t* allocated = (size_t*)::operator new(sizeof(size_t) + \n5 * sizeof(A));

*allocatedArea = 5; // store size
NewTrick Temporary;
A* fiveAs = NewTrick::operator*<A>(&Temporary, 
(A*)(allocated + 1));
```

That means in NewTrick::operator* a different address will be passed to SetAllocationType than the one passed to RegisterAllocation from within ::operator new[]. Let us consider our possibilities:

- Adjust the pointer in ::operator new[] before calling RegisterAllocation. Without type information we have no chance to determine if the adjustment is really necessary. Arrays of built-in types, like int, will not be prefixed with size.
- Do the necessary pointer adjustment when calling SetAllocationType. Using the described syntax we are not aware of whether it was a single object or an array allocation.

The safe solution is either having fiveAs (see Figure 7.2.) available in ::operator new[], or having allocated available in NewTrick::operator*. The first is impossible because fiveAs is calculated only after ::operator new[] returned. With the current syntax we can only pass the value via global variables. That pops up multithreading issues, and it is also not guaranteed that there are no subsequent calls to ::operator new[] before the corresponding NewTrick::operator* gets executed. We should probably use a stack on each thread. Do not forget that NewTrick::operator* cannot differentiate between new and new[], therefore we should do this extra pointer administration also in the more frequent single object allocation case. Though this solution could probably work, we rather chose a simpler one and gave up the nice syntax. Instead of

```cpp
A* fiveAs = new A[5];
```

one will have to write
CHAPTER 7. TYPE-PRESERVING HEAP PROFILER FOR C++

```cpp
1 template<class T> T* New_(int Count) {
2     void* allocated;
3     T* result = new (&allocated) T[Count];
4     SetAllocationTypeAndCount(allocated,
5         /* result, */ // optional
6         &TYPE_IDENTIFIER<T>::mDummy, Count);
7     return result;
8 }
9 void* operator new[](size_t size, void** pAllocated) {
10    return *pAllocated = RegisterAllocation(malloc(size));
11 }
```

Listing 7.3: Obtaining pointer to allocated area in array allocations.

```cpp
A* fiveAs = New_<A>(5);
```

for array allocations. `New_` passes a pointer to a local stack variable (named `allocated` in the previous example), to `::operator new`. The operator is supposed to write the actual allocation address into this variable, see Listing 7.3. Here `RegisterAllocation` returns its argument unchanged allowing the elimination of a temporary variable. Note that passing `allocated` as `void*` would have conflict with placement `new`. Be careful to add this code to a place where the `new` macro has not yet been defined. We also pass `Count`, which eliminates the need of knowing `sizeof(T)` when the allocations are analysed. In fact, having `Count` at hand, we will use it for the opposite purpose, to calculate `sizeof(T)` based on the allocation sizes.

Fortunately array allocations are less frequent, hence the introduced syntactic overhead hurts less. Regardless of how frequently the new form of array allocation is advertised among the coders, likely they will sometimes forget not to use the standard form. In those cases `SetAllocationType` will get an address that is not present among the allocation entries. It is a silent killer, because it either produces a run time error if the execution once wanders there, or just simply distorts the measure. We should somehow protect the framework against this kind of misuse. Using the standard form of `new[]` should result in a broken build. Though we cannot delete the always existing `void* operator new[](size_t)` overload, we already have a define for `new`. Let us add an extra parameter to it:

```cpp
struct UseNew_ { static UseNew_ st; };
#define new NewTrick() * new (UseNew_::st)
```

Of course this will also affect the single object allocations, we should modify the declaration of `operator new` accordingly:

```cpp
void* operator new(size_t size, const UseNew&) { ... }
```
For the `new[]` case let us declare an overload that does not match:

```cpp
void* operator new(size_t size, const UseNew_*);
```

With some compilers we would get compilation errors wherever someone tried to use the standard `operator new[]` syntax. With other compilers, however, we will get no errors, they just silently fall back on using the standard one. On these compilers it is better to use the given overload without implementation:

```cpp
void* operator new(size_t size, const UseNew_*);
```

This will result in an unresolved external at link time. The exact place of misuse is not revealed, but at least the build gets broken. In most cases a clever regular expression is enough to find the problematic line.

### Additional variants of `new`

We have discussed the two most common usages of `new`: the single object and the array forms. There, however, can be arbitrary additional overloads with their special custom behaviour. An important example of that is placement `new`, which is the official way of explicitly calling the constructor on a previously allocated memory area:

```cpp
char Mem[sizeof(std::vector<int>)];
new (Mem) std::vector<int>();
((std::vector<int>*)Mem)->push_back(42);
```

With the first version of our `new` macro, this code would compile, but the construct would be treated as an allocation and would silently result in corrupt allocation administration. The current version, however, produces the following code:

```cpp
new NewTrick() * new (UseNew::st) (Mem) std::vector<int>();
```

which is ill-formed [21]. In fact in these cases we should not use our macro. The proper solution is turning it off for these special lines:

```cpp
#pragma push_macro("new")
#undef new
new (Mem) std::vector<int>();
#pragma pop_macro("new")
```

The undefinition of `new` must be as limited as possible. Even undefining it only for that single line can lead to administration errors if the parameter expressions of the constructor call contain another `new`. The recommended style breaks these calls into two lines:

```
new NewTrick() * new (UseNew::st) (Mem) std::vector<int>();
```
#pragma push_macro("new")
#undef new
new (Mem2)
#pragma pop_macro("new")
    std::auto_ptr<int>(new int(42));

push_macro and pop_macro are not part of the C++ standard [21], but most
compilers support them. The simplest standard compliant solution would simply
undefine the macro and redefine it afterwards. That is, however, against the
DRY principle as the definition of new would be repeated at each location. We
recommend putting its definition and the corresponding undefinition to dedicated
headers:

#include "undef_new.h"
new (Mem2)
#include "define_new.h"
    std::auto_ptr<int>(new int(5));

It is important for these files not to contain header guard as they can be used
multiple times in the same compilation unit.

typedefs Array typedefs can make our lives harder too. Consider the following
example:

typedef A HideArrayA[10];
A* Array = new HideArrayA;

Despite the allocation seems correct, it is actually a call to operator new[]. In
these cases we have to substitute the typedef, which unfortunately hurts the DRY
principle (see 1.1.1).

Containers Having all these implemented, we started using our heap profiler
and the type information seemed correct. Sometimes, however, the occupied mem-
ory was not matching our knowledge about the software. It turned out that alloca-
tions of STL containers [20] were all typeless, because they internally used malloc.
To solve the problem either std::allocator should be modified, or a custom al-
locator should be used. The first is simpler if a custom STL implementation is
already part of the code base, but sometimes not possible. The custom allocator
probably produces another syntactic overhead when using STL containers. STL
is only an example, in fact each generic data structure probably uses malloc for
allocation.
### 7.4. IMPLEMENTATION DETAILS

**Raw allocations**  Similarly, even at places outside of templates, we can encounter usage of raw allocation routines, typically together with placement new. Such behaviour is mainly driven by efficiency claims. In these situations our framework will not be able to produce relevant type information. We can get around this problem by adding explicit calls to `SetAllocationType`/`SetAllocationTypeCount` manually directly after the raw memory allocation. Moreover, even defining new pseudo types exclusively for type identification makes sense. That way symbolic names can be used at places where buffers of built-in types are allocated:

```c
struct MEM_TYPE_IMGBUF {};  
int* ImgBuf = New_<int,MEM_TYPE_IMGBUF>(w*h);  
// or  
int* ImgBuf = LogMemoryType((int*)malloc(w*h*sizeof(int)));  
```

where `LogMemoryType` is a template function:

```c
template<class T> T* LogMemoryType(T* Ptr) {  
  SetAllocType(Ptr, &TYPE_IDENTIFIER<T>::mDummy);  
  return Ptr;  
}
```

### 7.4.4 Call stack

With the previously described allocation hook method we are able to keep track of all heap operations of the software and maintain a database of the actually used memory areas. The entries by now can contain type, size, count and allocation timestamp. In this subsection I describe the details of collecting information about the call stacks. Unfortunately there is no standard way in the C++ language to obtain the actual call stack. Thus we have only the following suboptimal solutions:

- Use a platform dependent system routine. Quite often such a backtrace function simply does not exist or is not accessible from our programming environment. Even if we have a backtrace function available, usually its slowness [219] makes it unusable for our purposes.
- Write own version of a backtrace function. This is not impossible, but a self-made solution is very sensible for special optimisation techniques, and requires deep knowledge of the actual architecture. This approach is definitely against our platform independent intentions as it introduces numerous platform dependent techniques.
- Continuously maintain a vector of code addresses representing the call stack. This can be achieved by utilising instrumentation functions.
CHAPTER 7. TYPE-PRESERVING HEAP PROFILER FOR C++

The last option can be implemented so as to fulfil the necessary speed requirements while keeping compiler dependent parts on minimum.

Many compilers provide options for generating instrumented code for profiling purposes. If the given compilation option is set then dedicated enter and exit functions are called at the beginning and end of each function respectively. Table 7.1 shows the signatures of these special functions in different compilers [220]. We should avoid calling any instrumented function from within these hooks to avoid recursion. Some compilers provide syntactic elements for disabling instrumentation of particular functions. In case of Visual C++ we can choose from two instrumentation methods: to place the special calls inside the called functions (callcap), or around the call instruction at call site (fastcap).

We can maintain the call stack of each thread by adding and removing an entry to and from a thread local array in the enter and exit functions respectively. In normal operation the simple push and pop implementation is sufficient, but there are some rare scenarios when special handling is needed. In case of longjmp or exceptions the caller address passed to the exit function may differ from the one passed to enter. To handle C++ exceptions we have to find the address in lower positions of the array and pop the unwound elements.

A reference to the CCF node representing the last function of the call stack is stored in a newly registered allocation entry. For each heap allocation the CCF

<table>
<thead>
<tr>
<th>Compiler</th>
<th>enter function</th>
<th>exit function</th>
<th>how to disable</th>
</tr>
</thead>
<tbody>
<tr>
<td>g++</td>
<td>void __cyg_profile_func_enter (void *this_fn, void *call_site);</td>
<td>void __cyg_profile_func_exit (void *this_fn, void *call_site);</td>
<td><strong>attribute</strong>((<strong>no_instrument_function</strong>))</td>
</tr>
<tr>
<td>Intel C++ on IA-32 and Intel®64 PathScale PGI</td>
<td>void __cyg_profile_func_enter (void **this_fn, void *call_site);</td>
<td>void __cyg_profile_func_exit (void **this_fn, void *call_site);</td>
<td><strong>attribute</strong>((<strong>no_instrument_function</strong>))</td>
</tr>
<tr>
<td>Intel C++ on IA-64</td>
<td>void __cyg_profile_func_enter (void **this_fn, void *call_site);</td>
<td>void __cyg_profile_func_exit (void **this_fn, void *call_site);</td>
<td><strong>attribute</strong>((<strong>no_instrument_function</strong>))</td>
</tr>
<tr>
<td>Visual C++</td>
<td>void _CAP_Start_Profiling (void <em>call_site, void</em> this_fn);</td>
<td>void _CAP_End_Profiling(void *call_site);</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 7.1: Signatures of enter and exit functions in different compilers.
node matching the actual call stack has to be looked up or constructed in the CCF data structure (see 7.3.2). To keep speed overhead low, this operation has to be fast. Once a CCF node is found for a call stack element, it is worth memorising it in the given call stack function entry. That data is valid until the entry is removed from the stack. With this optimisation the lookup can remain relatively fast even if the call paths tend to be long.

Another potential optimisation is looking for a given child in the tree starting at the last hit at that level, assuming that there are subsequent calls to the same function from the call site. I recommend using a leftmost-child-right-sibling representation in CCF with fixed node size and linking the last child to the first, thus producing a circular list. This circular list of child nodes makes their iteration starting from the remembered last hit easier: follow next until the starting node is reached again. Apart from being an optimal storage of many call stacks, CCF serves room for storing summarised information at its nodes. Figure 7.3 shows the relationships of our three data structures: allocation entries, call stack and CCF.

Our framework has been implemented along the above lines, and has been successfully used in the following compilation environments:

- Visual C++ / Windows XP, 7
- Visual C++ / Windows CE
- g++ / iOS
- g++ / QNX
- g++ / android

Though applications for iOS are usually written in Objective C, in our case the bulk of the code is in C++, only a thin platform layer deals with the native API. The same holds for android, where applications are usually written in Java.

7.5 Usage

In the previous section I presented a method for collecting stack trace and detailed type information on heap operations. We also got familiar with the recommended data structures behind such a framework. Now let us see how we can extract the gathered data from a running application.

The profiling process consists of the following steps:

1. Profiler integration
2. Profiler configuration
3. Compile the profiling enabled application
4. Run the application, test different scenarios
5. Convert resulting profiler output dump files
void f1() {
    AddA(new A);
    f11(), f12();
    AddBMulti(5, New_<5>(B));
}

void f2() {
    f21(false);
    f21(true);
    f212();
    f211();
}

void f21(bool b) {
    if (b)
        f212();
    else
        f211();
}

Figure 7.3: The connection between allocation entries, call stack and CCF. Dotted lines in the tree denote physical-only links, while the dashed ones are logical-only. The figure shows the moment right before the last call stack entry is updated to point to f211. The last entry currently points to f212 as a result of the preceding call. lastValidLink will be set to 3 afterwards.
6. Analyse/visualise results

We have discussed the tasks of step 1 in Section 7.4.

In many cases we do not want to measure the whole run of the application, instead we may want to restrict the scope of profiling into a time interval. To be able to define this range either at compilation time, or initiated by user actions at run time, we have to add necessary support code. There are three different approaches:

- Profile the whole run.
- Start and end profiling programmatically with explicit calls to the framework from client code. The free operations should handle the cases when the allocation is not yet present in the administration.
- Start and end profiling at run time. For this we have to add the trigger functionality to the client code. It can be a keyboard shortcut, a special mouse gesture, the presence of a file with a specific name (non-gui applications), a special http request (network server application) etc.

Every feature of the profiler has its own cost. We pay for them in seconds and/or kilobytes. It is rare that for our specific measurement we need all the data the profiler can collect. I recommend implementing them as orthogonal as possible enabling the user to freely select the desired components. In a software environment that uses garbage collection techniques one may want to perform garbage collection right before creating the dump not to be misled by already (logically, not physically) freed elements in the dump. On the contrary, one may specifically want to dump a state right at the point when garbage collection is to be performed to see what memory state has triggered it. These alternatives are usually controlled by preprocessor directives and/or configuration files which are read typically at program startup. In the configuration step these defines and configuration files are set up according to the needs of the given measurements. Do not forget to enable map file generation for the build process, it will be required later for the types and call stacks.

In the rest of the chapter I describe how a concrete implementation works which makes the discussed method more tangible. Note that though a specific implementation is shown, we are still speaking about a general approach that has to be tailored to the actual environment. The presented framework is an example of such a tailored version.

Once we have our application armed with the configured profiler code we can perform the test scenarios. They vary from simple ad hoc operations driven by curiosity (called exploratory testing [221]) to well defined test cases of scalability, a concrete bug of out of memory condition etc. Sometimes dumps of separate runs
are subject to comparative analysis. Depending on the chosen components the following types of dumps are generated:

- Regular dumps of freed elements. Whenever the framework exceeds its limit of storing freed allocation entries, it flushes them to a file.
- Full dump of the allocation entries, the freed elements and the CCF at exit. Non-freed elements of this dump correspond to leaked memory areas.
- On demand dump of the CCF, the allocation entries and optionally the freed elements.

The naming of profiler output files follow a numbering convention to handle multiple dumps properly. In our implementation all these files are plain text files. See Listing 7.4 for a fragment of a dump.

```
Listing 7.4: Text file listing allocation entries.
```

```
1 HeapLog|version=4|address|size|count|type_id|tick|
       freed_tick|ccf_node|alloc_type|nomansland=0,0
2 22A19F00,32,8,0073D411,132,132,022B7864,0
3 22A19EC8,32,8,0073D411,132,132,022B7864,0
4 0212DF10,30,30,0073D3AD,132,132,022C68C4,0
5 298D0B58,30,30,0073D42E,132,132,022C68DC,0
6 298D0B90,30,30,0073D3AD,132,132,022C68C4,0
7 22A19E30,124,31,0073D411,132,132,022C68F4,0
8 ...
```

In some systems additional memory is allocated to provide bound checking [222]. Such additional memory area, often called no man’s land [223], require special handling. No man’s land information is also added to the header. Nomansland values are positive if and only if bound checking is enabled in our memory management. When bound checking is enabled extra bytes are allocated at both ends, and are filled with specific bytes. At deallocation the bound checker framework examines if no man’s land bytes still hold these specific values. If not, there was presumably either a buffer overrun or an underrun on that block overwriting them. The call stack information in the allocation entry can help us finding the possible causes. In rare cases the corruption has nothing to do with the given block, but is a result of a ‘random’ memory write through a dangling pointer. Double delete problems, where an already properly deallocated area receives another deallocation call can easily lead to such random writes if the given already deallocated area has been assigned to another object in the meantime. Then the destructor of the previous
7.5. USAGE

```cpp
struct A {
    B* mB;
    ~A() { Clear(); }
    void Clear() {
        delete mB;
        // missing: mB = nullptr;
    }
};
```

Listing 7.5: Code producing double delete.

```plaintext
GraphLog|version=4|columns=allocations,allocatedBytes,
maxAllocatedBytes
> >
> >
4,107856,107856
<0054CDA0,02261024
4,107856,107856
<0054A7A0,0226100C
> >
> >

Listing 7.6: CCF dump.
```

object runs on the newly allocated object of the same or a different type. See 7.5
for an example code producing double delete.

The `ccf_node` field of the dump if the raw value of the CCF node pointer. It
may sound strange that we write live pointers of run time structures into an off
line dump, but this is actually a very convenient way to connect the allocation
entries with the CCF. CCF dumps contain the `this` pointer of CCF nodes and
thus we can easily associate them with the allocation entries. The format of the
CCF dump has a header similar to that of the allocation entry dump. It has a
version tag and the description of columns. The columns describe what kind of
summarised information is stored in the nodes of the CCF. We get the dump file
from a preorder traversal. A `>` character is written to the output when we visit
a node, and a corresponding `<` character is written when we leave the node. The
summarised information of the actual node is written before leaving. The function
address the node represents and the pointer of the node itself is written after the
`<` character, see Listing 7.6.

Once we went through our test cases we have dozens of these dump files. Their
format is suitable for being dumped from the application easily, but they are not
appropriate for direct analysis. We have to apply our converter tool to generate
one or more .muar files from these dumps. The extension stands for memory usage
analyser result, it is the base format of our visualiser application, see Section 7.6.

allocation entry dump
freed entry dumps \text{convert} \rightarrow .muar file

This .muar file is actually a sequence of enhanced allocation entries. The entries
in this file contain call stack information, thread id, can have colours etc. Same
call stacks are not repeated in the file to save space, see Listing 7.7. To resolve the

functions in the call stacks and the types the corresponding map file is necessary.

The presented profiler framework is general enough to be capable of tracking
the allocations of any resource, not only allocations made through the \texttt{new}
operator. Small object allocator \cite{129} is a technique for optimising memory usage
performance in systems where small allocations are frequent. Small object allo-
culators serve small allocation requests from preallocated bigger pools. Without
special handling the profiler could track only the allocations of these pools or the
small objects in them depending on where the allocation registration is attached.

By adding an allocation type member to the allocation entry structure these
different kinds of allocations can be distinguished and data can be collected for
both. There are further allocation types such as video ram, reference and marker.

\begin{Verbatim}[numbers=left]
1 ... 17 0x00512F80
2 callstack_at_alloc=16 18 0x005161C0
3 pointer=0x2294D740 19 0x004A83A0
4 count_at_alloc=1 20 0x004BC810
5 size=32 21 0x77618B9A
6 tickcount_at_alloc=5179 22 0x004B4090
7 alloc_type=0 23 0x00574C80
8 24
9 callstack_at_alloc=8 25 callstack_at_alloc=88
10 pointer=0x2298D740 26 pointer=0x229AD750
11 count_at_alloc=1 27 count_at_alloc=212
12 size=32 28 size=1272
13 tickcount_at_alloc=5195 29 tickcount_at_alloc=3190
14 alloc_type=0 30 type=0x005863FD
15 31 alloc_type=0
16 callstack ID = 88 32 ...
\end{Verbatim}

Listing 7.7: Muar file example.
When manual reference counting [224] is used to implement shared ownership there is a chance for the reference count to go out of sync. As normally only the allocations are tracked by the profiler, we can only find the first reference to the leaked object but not all further references. Similarly the individual release operations are not tracked either. When the reference leak hunter component is enabled in our system, allocation entries are created whenever a reference counted object is referenced or released, storing the corresponding call stacks and times. These entries form a chain, and all of them are deleted from the administration when the reference count reaches zero and the object is freed. Otherwise detailed information is available for finding the unwanted reference increment or the missing decrement. The marker allocation type allows registering certain program events among allocation entries. These entries are visualised in the analyser tool which helps analysing heap behaviour according to the given program events. In the converter tool any combination of allocation entry types can be specified to customise what kind of entries are to analyse.

7.6 Visualisation

Profiling industrial sized applications can produce enormous amounts of data. Understanding this raw information is often very time consuming and can leave crucial connections unrevealed. In the last decades visualisation tools proved to be key tools for program comprehension [225]. These tools provide an extremely good overall picture about the internal processes of applications. Quite often, however, we want to examine only a special subset of the data. Such subset can be the memory usage associated to a certain data structure, type or thread for example. Users would prefer to define their own specific views of profiling information. To support this request we provide elementary processing steps (filters) on the allocation entries and a way to combine these elements into complex queries as filter graphs [226].

Our visualiser tool (memory usage analyser) works with an ordered sequence of coloured allocation entries. These entries have the following properties: address, size, count, type (address of TYPE_IDENTIFIER<type>::mDummy), call stack (function addresses), time at allocation, time at free, thread id, colour.

Some of the fields may be missing from our input data depending on the configuration (see 7.5). For types and call stacks to appear properly we always need the corresponding map file.

We provide a visual language for constructing the filter graph, where boxes represent filters, and edges represent data flow. The user can define complex queries using graphical interface via drag and drop technique. These graphs represent different measurement patterns and can be stored and reused later. Figure 7.4
Chapter 7. Type-Preserving Heap Profiler for C++

shows how a filter graph appears in our tool.

![Filter Graph Diagram]

Figure 7.4: A complete filter graph. Each box has its inputs on the left, and its outputs on the right. Straight lines between input and output pins denote connections.

Boxes in these filter graphs can execute input or output actions, filter on the values of given fields (including timestamp, call stack, type etc.). Aggregations like average or sum field values can also be performed with boxes. The colouring box changes the colour attribute of the passing entries to customise visual appearance. With this general approach we can build the following queries, for example:

- keep allocation entries invoked from within some `std::vector` instance
- colour entries whose type begins with `std::` to blue
- keep allocations which lived (no) longer than a given time
- keep allocations only of a given thread

Allocation entry sequences (possibly already processed with filter graphs) can be visualised graphically in different views. The timeline view gives a quick overview on how total memory usage changes over time, see Figures 7.7 and 7.8. If the entries have different colours, we see them stacked. The stacked layers can be freely reordered. By hovering over the chart the details of the actual time slot are presented.

In memory view (Figure 7.5 we can explore a grid representation of the physical memory. By choosing alternating colours we can easily differentiate separate entries, otherwise the colour tags of the entries determine pixel colours. We can define how many bytes one pixel represents. Hovering over the pixels the details of the actual allocation entries appear.

Text view is useful for going through the allocations one by one, typically after a filter procedure. Some additional operations we can perform on our data set:
7.6. VISUALISATION

Figure 7.5: Memory view in Memory Usage Analyser displaying pool allocation entries of an application using small object allocator (see 7.5) and the objects in the pools respectively, depending on the filter applied during conversion (see 7.5. Alternating colors clearly show block boundaries.

- join entries of same type/call stack
- unify the leafs of the CCF until the sum of allocation size reaches a specified limit
- sort entries by one of their fields (stable sort \[227\] to support multifield sort)
- calculate the differences and common parts of two sets

Some of the frequently used complex operations performed in the analyser:

- keep entries allocated or deallocated in a given time range
- join by type, then sort descending by occupied memory
- colour entries of a given type, or instances of a given template, then consult timeline view
- switch to memory view with alternating colours to check fragmentation and pool behaviour

7.6.1 Algorithmic challenge behind timeline view

Timeline view is a powerful tool for visualising trends in the heap behaviour of the examined application. This view is useful both for detailed analysis of a specially filtered set of allocation entries and for a quick overview of total program behaviour. Every column of the timeline view represents a fixed length time slice of the run.
From implementation point of view we need a quick way of determining the active set of allocation entries in each time slice. With the most usual setting of one pixel wide columns the number of time slices \(= n_t\) can be more than 1000. In a longer heap history the number of allocation entries \(= n_a\) can easily reach millions. Tables D.1, D.2 and D.3 contain real examples of view properties.

Let \(E\) denote the space of allocation entries, and \(M\) the set of valid entries. An \(M \in M\) is an entry set, the abstraction of a mauer file (see 7.5).

\[
N^\infty := N \cup \{+\infty\} \\
E := \mathcal{T}_{\text{alloc}} \times \mathcal{P}_{\text{type}} \times N \times N \times N \times N \times N \times N \times N \times \text{COL} \\
M := \{e \in E | e_{at} \leq e_{ft}\}^*
\]

**Definition.** Timeline function of an entry set

\[T : M \to (N \to M)\]

such that

\[T(M)(i) = \{m \in \text{Im} M | [m_{at}, m_{ft}] \cap [g + i \delta, g + (i + 1)\delta) \neq \emptyset\} \quad (M \in M, \quad T^M_i := T(M)(i) \quad i \in N)\]

where \(\delta\) denotes the fixed length of the time slices and \(g\) (genesis) the first allocation time:

\[g := g(M) := \min_{m \in \text{Im} M} m_{at} \quad (M \in M)\]

while

\[\omega(M) := \max \{i \in N | T^M_i \neq T^M_{i+1}\} \quad (M \in M)\]

is the last time slice index.

Now we can define \(n_a\) and \(n_t\) formally:

\[n_a(M) := |\mathcal{D}_M| \quad (M \in M)\]
\[n_t(M) := \omega(M) + 1 \quad (M \in M)\]

**Definition.** Duration of an entry set

\[d(M) := \max ((\text{Im} M_{ft} \cap N) \cup \text{Im} M_{at}) - g(M) \quad (M \in M)\]
7.6. VISUALISATION

Algorithm 7.1: Unoptimised version of time slice processing.

\[
\begin{align*}
g &\leftarrow \min_{m \in I M} m_{at} & // O(n_a) \\
h &\leftarrow \max_{m \in I M} m_{ft} & // O(n_a) \\
\omega &\leftarrow \left\lceil \frac{h-g}{\delta} \right\rceil + 1 \\
\text{for } i = 0 \text{ to } \omega \text{ do} & // O(n_t) \\
\quad \text{for } m \in I M \text{ do} & // O(n_a) \\
\quad \quad \text{if } m_{at} < g + (i+1)\delta \land m_{ft} \geq g + i\delta \text{ then} \\
\quad \quad \quad \text{addtoslice}(i,m)
\end{align*}
\]

To display the timeline view, the visualiser should determine and process the allocation entries of each \(T_i^M (i \in [0..\omega(M)])\) time slice. This processing step appears in the presented algorithms as a call to \text{addtoslice}(i,m) \( (m \in T_i^M)\). An unoptimised version would need \(O(n_t \cdot n_a)\) operations, see Algorithm 7.1.

Let \(X_{f \circ f}(y) := \{x \in X \mid x \cdot f \ast y\} \quad (\ast : X \times X \rightarrow \mathbb{L}, y \in X)\) be a shorthand notation for subsets defined by relations. From the definition of \(T\) follows that

\[
T_i^M = (I M M)_{at \leq (g + (i+1)\delta) \setminus (I M M)_{ft \leq (g + i\delta)} \\
= (M \setminus (I M M)_{at \geq (g + (i+1)\delta) \setminus (I M M)_{ft \leq (g + i\delta)}})_{at} \\= F_i
\]

That is, \(T_i^M\) contains entries that were allocated before \(g + (i+1)\delta\) and were freed after \(g + i\delta\). Furthermore,

\[
A_i \subseteq A_{i+1} \land F_i \subseteq F_{i+1} \land F_i \subseteq A_i. \tag{7.2}
\]

Let us sort the entries by their allocation and free times and denote the resulting orders as \(\sigma_{at}\) and \(\sigma_{ft}\) respectively \((\sigma_{at}, \sigma_{ft} : [1..n_a(M)] \leftrightarrow [1..n_a(M)])\):

\[
i < j \Rightarrow M(\sigma_{at}(i))_{at} < M(\sigma_{at}(j))_{at} \\
M(\sigma_{ft}(i))_{ft} < M(\sigma_{ft}(j))_{ft}
\]

Then

\[
A_i = (M \circ \sigma_a) [[1..\alpha_i]] \\
F_i = (M \circ \sigma_f) [[1..\phi_i]] \tag{7.3}
\]

for some appropriate \(\alpha_i, \phi_i \in \mathbb{N}^+\). Algorithm 7.2 is an optimised version for calculating time slices based on 7.1, 7.2 and 7.3.

127
Algorithm 7.2: Optimised time slice calculation.

\[
\begin{align*}
    a_0 & \leftarrow [1 \ldots n_a] && \text{// } O(n_a) \\
    o_f & \leftarrow [1 \ldots n_a] && \text{// } O(n_a) \\
    \text{sort } a_0 \text{ by } M_{at} && \text{// } O(n_a \log n_a) \\
    \text{sort } o_f \text{ by } M_{ft} && \text{// } O(n_a \log n_a) \\
    r & \leftarrow 0 \\
    S & \leftarrow M \circ o_a \\
    \text{for } i = 0 \text{ to } \omega \text{ do} && \text{// } O(n_t) \\
    \quad S' & \leftarrow S \setminus (\notin F_i \setminus F_{i-1}) && \text{// remove } (M \circ o_f)[[\phi_{i-1} + 1 \ldots \phi_i]] \\
    \quad r & \leftarrow r + |D_S| - |D_{S'}| && \text{// number of removed entries} \\
    \quad S & \leftarrow S' \\
    \quad \text{for } j = 1 \text{ to } \alpha_i - r \text{ do} && \text{// } O(n_d) \text{ total} \\
    \quad \text{addtoslice } (i, S(j))
\end{align*}
\]

In its current form Algorithm 7.2 is too abstract for determining its time or space characteristics. \( n_d \) denotes the total number of processed allocation entries in the timeline view. Let us define a subset type \((S, \{\text{subset, } \setminus, \text{begin}\})\) and its iterator type \((I, \{\text{iter, atend, index, next}\})\) for storing and referencing entry indices as follows:

- \( \text{subset} : \mathbb{N}_{size} \rightarrow S \) creates a full subset of size elements: \([1 \ldots size]\)
- \( \setminus : S \times \mathbb{N}_{index} \rightarrow S \) removes index from the subset
- \( \text{begin} : S \rightarrow I \) iterator to first element

- \( \text{iter} : S \times \mathbb{N}_{index} \rightarrow I \) creates iterator of \( S \) pointing to \( \text{index} \)
- \( \text{isvalid} : I \rightarrow \mathbb{L} \) iterator has not passed all elements
- \( \text{index} : I \rightarrow \mathbb{N} \) the pointed index in the subset
- \( \text{next} : I \rightarrow I \) step to the next element of the subset

Algorithm 7.3 shows the same optimised algorithm now using the \( S \) and \( I \) types.

Algorithm 7.3 can be divided into preparation (P) and actual visualisation (V) sections. An initial preparation cost is acceptable in practice if in exchange we get a more fluent user experience when interacting with the software. In timeline view the user typically wants to adjust \( \delta \) and other display parameters like scaling (how much memory one pixel should represent), the order of colours, or the time
7.6. VISUALISATION

Algorithm 7.3: Optimised time slice calculation using the subset and iterator types.

P: $o_a \leftarrow [1 \ldots n_a]$  // $O(n_a)$ 
    $of \leftarrow [1 \ldots n_a]$  // $O(n_a)$ 
    sort $o_a$ by $M_{at}$  // $O(n_a \log n_a)$ 
    sort $of$ by $M_{ft} \circ o_a$  // $O(n_a \log n_a)$

v: $S \leftarrow \text{subset}(n_a)$  // $O(n_a)$ 
    $k \leftarrow 0$
    while $k < n_a$ do
        // remove elements freed in $T_{i-1}$
        while $k < n_a \land (M_{ft} \circ o_a)(of(k)) < g + i\delta$ do
            $S \leftarrow S \setminus of(k)$  // $O(1)$
            $k \leftarrow k + 1$  // $O(n_a)$ total
        // process $T_i$
        $I \leftarrow S$.begin ()  // $O(1)$
        while $I$.isValid () do  // $\text{isValid}:O(1)$
            $m \leftarrow (M \circ o_a)(I\.index ())$  // $O(1)$
            if $m_{at} \geq g + (i + 1)\delta$ then
                break
            addtoslice($i,m$)  // $O(n_d)$ total
            $I \leftarrow I\.next ()$  // $O(1)$

window (what time period to display instead of a full view). For these interactions only $V$ has to be rerun, consequently the focus is on making $V$ fast, while keeping $P$ at an acceptable level.

We can see that Algorithm 7.3 has $O(n_a + n_d)$ time complexity, provided that the subset and iterator operations run as indicated ($O(\text{size})$ for $\text{subset}$ and $O(1)$ for all others). The visualiser has to deal with each displayed entry at least once ($n_d$) in the given time slice and each entry has to be processed ($n_a$), therefore $time(V) = \Omega(n_a + n_d)$. This means our optimised algorithm is optimal. Note that the $\Omega(n_a)$ component in minimal complexity is valid only for full views. For partial views, where the displayed entries are prefiltred (displaying only a time range, or a certain type of entries, or by any other condition), $n_d > n_a$ does not necessarily hold and minimal time complexity drops to $\Omega(n_d)$. For those views the algorithm is suboptimal, though still much faster than the naive version. Real usage examples (see Table D.1) show that $n_d < n_a$ only rarely happens. Optimising visualisation of these rare subset of partial views is not scope of this discussion.
We made assumptions about the complexity of subset and iterator operations. Now I show a possible representation and corresponding implementation that fulfills these requirements while keeps memory footprint low. In this representation a fixed size vector of integers \( s \) stores the subset \( S \):

\[
 s : [0 \ldots \text{size} + 1] \rightarrow [0 \ldots \text{size}]
\]

with the following invariant:

\[
 \forall i \in S : \begin{cases} 
 s[i] = 0 \\
 n(i) \in S \lor n(i) = \text{size} + 1 \\
 p(i) \in S \lor p(i) = 0
\end{cases}
\]

where

\[
 p, n : [1 \ldots \text{size}] \rightarrow [0 \ldots \text{size} + 1] \\
 n(i) := i + 1 + s[i + 1] \\
 p(i) := i - 1 - s[i - 1].
\]

The full set is represented by an all zero vector: \( \{0\}_{i=0}^{\text{size} + 1} \). Figure 7.6 shows an example subset and its \( s \) vector representation.

![Figure 7.6: Representation of a subset as a vector.](image)

The iterator is an ordered pair of a reference (pointer) to the subset \( ps \) and an index (index) of a zero element in \( s \). Therefore index is either the index of an element of \( S \), or \( \text{size} + 1 \):

\[
 (ps, index) \in P \times [1 \ldots \text{size} + 1] \\
 [ps][index] = 0
\]

where \( [ps] \) is the subset referenced by \( ps \). Algorithm 7.4 demonstrates how subset and iterator types can be implemented using this representation.

The operations are very simple, it is easy to see that they fulfil our time complexity requirements.
Algorithm 7.4: Implementation of subset and iterator types.

Function subset(size) // O(size)
    return [0 .. size + 1]

Function \((s,index)\) // O(1)
    \(k \leftarrow s[index - 1] + s[index + 1] + 1\)
    \(s[index + s[index + 1]] \leftarrow k\)
    \(s[index - s[index - 1]] \leftarrow k\)
    return s

Function begin(s) // O(1)
    return iter \((s,1 + s[1])\)

Function iter(s,index) // O(1)
    return (addressof(s), index)

Function isvalid(i) // O(1)
    return i.index > 0 \&\& i.index < size

Function index(i) // O(1)
    return i.index

Function next(i) // O(1)
    return (i.ps, i.index + 1 + s[i.index + 1])

So far we considered time complexity, but only mentioned that space complexity should be kept low. The described optimised algorithm with its subset and iterator implementations altogether require 3 extra integers:

\[
[1 \ldots n_a] \times [1 \ldots n_a] \times [1 \ldots n_a]
\]

for each allocation entry and O(1) additional space for the local variables and the two additional items in \(s\) (0th and \(size + 1\)st). An allocation entry originally occupies the space of 10 integers, therefore the space overhead of this solution is 30%. For having optimal timeline view speed this space overhead is acceptable in practice.

The way I defined the timeline view problem, namely the algorithm has to call \texttt{addtoslice} for each displayed entry, is a rather general approach. On the one hand generality allows us to implement any kind of actual visualisation and calculation of statistics of time slices on top of the described optimal algorithm, but on the other hand the algorithm cannot exploit the custom properties of the application to achieve even better performance.

With the \texttt{addtoslice} approach neighbouring time slices have to build up from scratch even if they are almost or totally the same. Many actual display applications does not require this level of generality, and can also be implemented easily
if we expect them to work only with the differences between neighbouring time slices. Let us see, for example, the task of displaying the sum of active memory entries in a time line view.

**Definition.** Memory usage chart of an entry set

\[ U : \mathcal{M} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \]

such that

\[ U(M)(i) = \sum_{m \in T_i^M} m_{\text{size}} \quad (M \in \mathcal{M}, i \in \mathbb{N}) \]

\[ U_i^M := U(M)(i) \]

**Definition.** Timeline difference functions of an entry set

\[ \Delta T^+, \Delta T^- : \mathcal{M} \rightarrow (\mathbb{N} \rightarrow \mathcal{M}) \]

such that

\[ \Delta^+ T(M)(i) = T^M_{i+1} \setminus T^M_i \]

\[ \Delta^- T(M)(i) = T^M_i \setminus T^M_{i-1} \quad (M \in \mathcal{M}, i \in \mathbb{N}) \]

\[ \Delta^+ T_i^M := \Delta^+ T(M)(i) \]

\[ \Delta^- T_i^M := \Delta^- T(M)(i) \]

where \( T' \) is the extension of \( T \) to support negative time slice indices:

\[ T' : \mathcal{M} \rightarrow (\mathbb{N} \rightarrow \mathcal{M}) \]

\[ T'(M)(i) = T^M_i \quad (M \in \mathcal{M}, i \in \mathbb{N}) \]

\[ T'(M)(i) = \emptyset \quad (M \in \mathcal{M}, i \in \mathbb{Z}^-) \]

\[ T_i^M := T'(M)(i) \quad (M \in \mathcal{M}, i \in \mathbb{Z}). \]

A call sequence \((C(M))\) visualises \( U(M) \) \((M \in \mathcal{M})\), if it consists of `addentry(m)` and `removeentry(m)` calls for time slice difference entries \((m \in M)\) preceded by and terminated by calls to `startslice()` and `finisheslice()` respectively:

\[ C(M) = \omega(M) \]

\[ \prod_{i=0}^{\omega(M)-1} c_i \quad (M \in \mathcal{M}) \]

where

\[ c_i = \{(1, \text{startslice}(0))\} \cup \{(1, \text{finisheslice}(\omega(M)))\} \quad (i \in [0..\omega(M)]) \]

with some

\[ d_i : [1..|\Delta^+ T_i^M| + |\Delta^- T_i^M|] \leftrightarrow \{\text{addentry}(i,m) | m \in \Delta^+ T_i^M\} \]

\[ \cup \{\text{removeentry}(i,m) | m \in \Delta^- T_i^M\}. \]
7.6. VISUALISATION

Algorithm 7.5: Visualising $U(M)$ in $O(n_a)$ time after preparation.

Function $\text{CalcU}(M)$

\begin{align*}
P: & \quad o_a \leftarrow [1 \ldots n_a], o_f \leftarrow [1 \ldots n_a] \quad \text{ // } O(n_a) \\
& \quad \text{sort } o_a \text{ by } M_{at}, \text{ sort } o_f \text{ by } M_{ft} \circ o_a \quad \text{ // } O(n_a \log n_a)
\end{align*}

\begin{align*}
v: & \quad k_a \leftarrow 0 \quad \text{ // indexing } M \circ o_a \\
& \quad k_f \leftarrow 0 \quad \text{ // indexing } M \circ o_a \circ o_f \\
& \quad i \leftarrow 0 \quad \text{ // time slice index}
\end{align*}

\text{starts\_slice}(i)

\begin{align*}
\text{while } & k_a \leq n_a \land k_f \leq n_f \text{ do} \quad \text{ // } O(n_a) \\
& \quad t_e \leftarrow g + (i + 1)\delta \\
& \quad t_f \leftarrow \begin{cases} 
(M_{ft} \circ o_a \circ o_f)(k_f) & \text{if } k_f \leq n_a \\
+\infty & \text{otherwise}
\end{cases} \\
& \quad t_a \leftarrow \begin{cases} 
(M_{at} \circ o_a)(k_a) & \text{if } k_a \leq n_a \\
+\infty & \text{otherwise}
\end{cases}
\end{align*}

\begin{align*}
& \quad \text{if } \min\{t_a, t_f\} \geq t_e \text{ then} \\
& \quad \quad \text{finish\_slice}(i) \\
& \quad \quad i \leftarrow i + 1 \\
& \quad \quad \text{starts\_slice}(i) \\
& \quad \quad \text{continue}
\end{align*}

\begin{align*}
& \quad \text{if } t_a \leq t_f \text{ then} \\
& \quad \quad \text{add\_entry}(i, (M \circ o_a)(k_a)) \\
& \quad \quad k_a \leftarrow k_a + 1
\end{align*}

\begin{align*}
& \quad \text{if } t_f \leq t_a \text{ then} \\
& \quad \quad \text{remove\_entry}(i, (M \circ o_a \circ o_f)(k_f)) \\
& \quad \quad k_f \leftarrow k_f + 1
\end{align*}

\text{finish\_slice}(i) \quad \text{ // here } i = \omega(M)

Function $\text{start\_slice}(i)$

\begin{align*}
U_i & \leftarrow \begin{cases} 
0 & \text{if } i = 0 \\
U_{i-1} & \text{otherwise (} i > 0 \text{)}
\end{cases}
\end{align*}

Function $\text{add\_entry}(i,m)$

\begin{align*}
U_i & \leftarrow U_i + m_{size}
\end{align*}

Function $\text{remove\_entry}(i,m)$

\begin{align*}
U_i & \leftarrow U_i - m_{size}
\end{align*}

Function $\text{finish\_slice}(i)$ \quad \text{ // nothing to do
Algorithm 7.5 visualises $U(M)$ in $O(n_a)$ time if $o_a$ and $o_f$ are already available from a preparation step. $O(n_a)$ time complexity is optimal with this set of calls (start-slice and add-removeentry), because every $m \in M$ entry should be processed once. Note that if there are many active entries on average and there are many time slices, then their product, $n_d$, can be substantially bigger than $n_a$. According to Table D.3 $n_d$ can be as high as 57,641,694,335 with $n_a$ being 143,222. Real usage examples (see Tables D.1, D.2, and D.3) confirm that high $n_d/n_a$ ratio is a frequent case, $>100$ ratios are not rare, even in partial views. In these typical cases Algorithm 7.5 outperforms Algorithm 7.3.

7.7 Usage examples

Reserved vector One could say type information is not that necessary, we have been working with our good old profilers for decades. Let us show an example where type had an important role. On embedded systems [228] there are much stricter memory limitations than on PCs. However, even these platforms have complex applications, like a real time navigation software for example. In our case the software had a special interactive visual element that tracked mouse movements (in fact touch movements) for processing the sequence at the end of or during the motion. This control used `std::vector` to store touch coordinates, then called clear on them when the data was no longer needed. At first glance this seems correct, but since the clear operation of `std::vector` does not free its allocated buffer [229], the corresponding memory block remained reserved. After using the software for a while hundreds of mouse events could accumulate in each such interactive component. When grouped the allocations by type, this event type was among the top 10 most memory consuming types of the whole application. It was responsible for ~400kB of the 12MB total memory consumption (~3.3%), which was too much for an unused reserve area. Without having type information about the allocations the memory wasted by these events could have been easily treated as necessary costs of the visual component.

In another case a functional component of a navigation system produced slowness, hang ups, but only in certain situations, never in in-house tests, which became a blocking issue of the project. After weeks of failed investigation attempts a memory logger enabled version was sent to field test to collect memory usage data. From the dumps we could easily identify a certain type that had much more instances than should have been, and that this number never decreased. The type was in direct connection with such a location specific behaviour that was not among our in-house test cases at the time. Again, it turned out that a `vector` could sometimes grow very big, and always occupied its biggest ever size.
A text editor  For demonstration purposes we have selected an open source C++ application, the Notepad++ free text editor [230]. This editor integrates Scintilla [231], a complex text editor component.

After integrating our framework first to Scintilla, then to Notepad++, we could get heap profile information at the end of each run. We opened some smaller and bigger files in Notepad++, and loaded the generated dumps in the visualiser. The timeline view showed that memory was allocated in bigger blocks as the file was being loaded. To determine what types are involved in the loading process and in representing the contents of the file, we applied the following operations: join by type, then sort descending by occupied memory. The result immediately showed that SplitVector&lt;char&gt;, char and SplitVector&lt;int&gt; are the most memory-consuming types. We assigned different colours to these types with a filter graph to examine dynamic behaviour. Figure 7.7 shows the resulting timeline view. The green area represents allocations of type char. It was strange that they were present during the loading process, but then they were deallocated. We restricted the view to a small time slice and from the call stack information we found which part of the code was responsible for those allocations. It turned out that they were the undo history. During loading each added buffer created an insert operation, that copied the string to the undo queue. This was done internally in Scintilla. After loading the file, Notepad++ cleared the undo history, hence the light green area disappeared. It was clear that maintaining the undo history when the insertions come from file loading wasted speed and memory. As a quick check we introduced a global flag for the file open situation and turned off undo handling for that special case. The resulting timeline view (Figure 7.8) shows that we suc-
ceeded in eliminating the unnecessary allocations. Without knowing the internal
details of the code, we could easily pinpoint a performance issue and could reduce
peak memory usage by 15%. This peak usage determines how big files we can be
opened in Notepad++. With this little modification we raised that limit by 18%.

7.8 Related work

The idea of adding extra arguments to \texttt{new} by defining it as a macro appears
in [232]. There \texttt{__FILE__} and \texttt{__LINE__} are passed to the custom overload allowing
for associating source code locations with allocations. With this technique a useful
memory leak finder tool can be easily implemented. Note that the file and line
information do not necessarily identify an allocation precisely:

\begin{verbatim}
vecA.AddA(new A(new B(a,b), new B(c,d)));
\end{verbatim}

There are many heap profilers on the market, providing a wide variety of fea-
tures from bound checking, memory leak hunting, call tree analysis etc. Most of
them are designed to be used on desktop computers, for middle sized applications.
When applied one of them to a strategic game with around 1GB memory usage,
the profiled version failed to launch. There were many heap allocations in the
software and their administration took more memory than the heap the program
used. By exceeding the 2GB heap allocation limit on the given platform heap
analysis was not possible. Though this limit could be raised to 3GB, one of the
3rd party components did not handle pointers with highest bit set correctly. We
had to give up using that tool for profiling.
Retrieving heap related type information is easier in languages where a reflection model is built into the language. Sometimes such profilers are part of the platform SDK [233, 234, 235, 208]. There is also a wide range of third-party solutions [236, 237, 238].

Memory Validator [239] of Software Verification Ltd. provides type information corresponding to C++ memory allocations. However, it prints the type information looked up from source code, rather than utilising the type system of the language itself. This approach requires debug information during run time execution and may fail in case of complex preprocessor macro definitions or intense code optimisations.

Valgrind [240] is an instrumentation framework for building dynamic analysis tools. It has a heap profiler module called Massif which does not provide type information on allocations. As the Valgrind framework is widely used, an alternative approach could be based on modifying Massif. However, Valgrind officially supports only the Linux and Mac OS X platforms.

GPTL [220] utilises instrumentation functions to perform performance analysis on different platforms. In [241] different visualisation methods of dynamic memory allocations are presented, including a similar one to our timeline view. In [235] stacked timeline views differentiating allocations by type are presented as an everyday profiling technique. Heapviz [209] visualises heap usage of Java programs as a graph. The idea of defining queries is envisioned in the future work section of that paper. The Leaks component in the Instruments profiler tool of the iPhone SDK [208] provides type preserving heap profiling for Objective C.

Hookit [242] is an extension of JPROF [243], a profiler for Java, to cover allocations in C codes as well.

MemTrack [244] is a memory profiler that uses a very similar approach that is presented in this chapter to capture type information. There typeid is used to retrieve the type name. Allocation information is placed directly as a hidden head of the allocations themselves, always allocating a bigger block. Source code location, size and type information are stored. There is room for improvement to add support for array allocations of types with non-trivial destructor, multithreading, call stacks.

For Chromium a modified version of clang was made to directly support tracking memory allocations with type information. After evaluating our method and MemTrack they decided to replace code generation for new and delete directly in the compiler. Compiling 
\[
A* a = new A(3,4);
\]
results in the following call:

\[
__op_new_intercept__(operator new(sizeof(A)),
sizeof(A), typeid(A))
\]
instead of `operator new(sizeof(A))`. `__op_new_intercept__` should be implemented by the user code and `__op_delete_intercept__` for `delete` similarly. They cite our ICSM’11 article in their design document [245].

C-strider [246] allows type-aware heap traversal of C programs by extending existing heap pointer graph traversal techniques with type information. The framework provides solution for the problem of untyped allocations, that are more common in C than in C++. Authors cite my publication as the only profiler they are aware of that provides fine-grained type information for C.

7.9 Contribution

Memory profilers are key tools for understanding the run time behaviour of modern object-oriented programs. Although languages like C++ strongly rely on types and classes, most memory profilers fail to provide sufficient information on actual types of memory allocations. I implemented a type preserving heap profiler for C++. The main contribution of my framework is the ability of providing detailed type information of each heap related operation additionally to usual profiling features. To support program comprehension a filter graph approach allows the user to construct arbitrary queries in a fairly convenient way. Various visualisation possibilities are available to examine profiling result. My solution is highly platform independent and has moderate memory and speed footprint. Timeline view is very useful in practice, but it needs efficient algorithms to cope with the huge amount of displayed entries. The presented algorithm is capable of handling millions of allocation entries and displayed entries well above $10^{10}$. As a case study I used the framework to find and fix a performance issue in a code base that was new and unknown to me.

**Thesis 5 (Type-preserving heap profiler)**. *I developed a method for preserving type information in C++ heap profilers. The method is platform independent, and has low memory footprint. I provided optimal algorithms for the performance critical timeline view visualisation. I demonstrated the applicability of the approach in real projects by pinpointing and fixing a performance bug in an open source text editor.*


### 7.9. CONTRIBUTION

<table>
<thead>
<tr>
<th>thesis name</th>
<th>relevant publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of unity build</td>
<td>•</td>
</tr>
<tr>
<td>Template metaprogram instrumentation</td>
<td>• • • • • • • • • • •</td>
</tr>
<tr>
<td>Methods for template metaprogram debugging</td>
<td>• • • • • • • • • • •</td>
</tr>
<tr>
<td>Methods for template metaprogram profiling</td>
<td>• • • • • • • • • • •</td>
</tr>
<tr>
<td><strong>Type-preserving heap profiler</strong></td>
<td>•</td>
</tr>
</tbody>
</table>
Chapter 8

Summary

My two key research areas are generic metaprogramming, especially C++ template metaprogramming, and working with large multiplatform C++ code bases in industrial environments. While both topics are covered by numerous academic papers and have many examples in the industry, there is a gap between them. The application of generic metaprogramming techniques tends to fail as practical industrial aspects come into play. 3 of my 5 theses directly focus on bringing the two worlds closer. My other two theses improve development efficiency in cases where standard methods are not longer sufficient at big scale. These other two theses implicitly support generic programming, in fact in those cases their effect is even more significant.

In the introduction chapter I described the most relevant notions and set the key focuses of my dissertation along with its structure. Thesis order followed the workflow of a program change. First the changed code has to be compiled with the build system (Thesis 1). Compilation of the specific units may employ generative metaprogramming techniques. For the diagnostics of our C++ metaprograms we may need TMP instrumentation (Thesis 2), a TMP debugger (Thesis 3) or even a TMP profiler (Thesis 4). After linking the object files to an application or service and optionally a set of run time loaded libraries, the generated code is loaded to memory and gets executed. Run time execution typically involves dynamic memory handling, which is achieved by heap operations (Thesis 5). In this order development efficiency improvement topics addressing build times and run time heap behaviour framed three theses on C++ template metaprogramming: instrumentation, debugging and profiling.

Both unity build (Thesis 1) and capturing types at allocations (Thesis 5) were already known tricks [70, 244]. I intentionally used the word “tricks” instead of technologies here. Their maturity was far behind the entry level of applying them in critical projects. The detailed analyses in the corresponding chapters are supposed to fill this gap. The pros, cons, advantages and limitations are clearly
enumerated, alternatives are shown, which all helps an architect making a go/no-go decision. Avoiding unwanted side effects and ensuring reliable behaviour also received strong focus in the discussions. Such detailed evaluations elevate these tricks to become techniques, which is my main contribution in Theses 1 and 5.

Template metaprogramming is a bit exotic topic and needs deep understanding of the compilation process and the syntax and semantics of generics. To define the context and establish a solid ground for the related chapters I devoted a separate chapter for generic metaprogramming. Then the focus narrows to C++ template metaprogramming in the three thesis chapters describing methods for instrumentation, debugging and profiling. Contrary to unity build and typed heap profiling here we cannot talk about existing approaches. The aim here was supplying working solutions to these specific needs. While diagnostic functionalities are natural parts of most programming environments, this was not true for C++ template metaprogramming. The numerous citations to my first research paper presenting the Templight approach indicate it was indeed a pioneer work.

TMP instrumentation can be used in itself for analysis, but most often it is rather a building block of higher level functionalities such as debugging or profiling. In the corresponding thesis chapters I presented methods that empower TMP with effective diagnostic capabilities. Even though these approaches are new, they perform well even in large systems and serve as a good foundation for further improvements. My main contribution in Theses 2, 3 and 4 is supplying solution, applicable methods and practices for the previously unsolved complex problem of TMP diagnostics.

The relevance of my research is demonstrated by citations both in academic papers and by developer communities. My heap profiler was chosen to be the opening meetup topic [15] of the Hungarian C++ Community [247]. TMP diagnostic functionalities are becoming official part of clang, the most modern C++ compiler as of today [185, 186, 184]. The impact of my research is summarised in Tables 8.1, 8.2 and 8.3.
CHAPTER 8. SUMMARY

| Thesis 1: Analysis of unity build                                      |   |
| Thesis 2: Template metaprogram instrumentation                      |   |
| Thesis 3: Methods for template metaprogram debugging                |   |
| Thesis 4: Methods for template metaprogram profiling                |   |
| Thesis 5: Type-preserving heap profiler                            |   |

<table>
<thead>
<tr>
<th>title</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>An Integrated Approach to Source Level Debugging and Compile Error Reporting in Metaprograms [248]</td>
<td></td>
</tr>
<tr>
<td>Identification of Idiom Usage in C++ Generic Libraries [249]</td>
<td></td>
</tr>
<tr>
<td>Identifying Programming Idioms in C++ Generic Libraries [250]</td>
<td></td>
</tr>
<tr>
<td>Type Checking C++ Template Instantiation by Graph Programs [251, 252]</td>
<td></td>
</tr>
<tr>
<td>Refactoring Feature Modules: Disciplined Generation of Reusable Modules [253]</td>
<td></td>
</tr>
<tr>
<td>Debugging Product Line Programs [254]</td>
<td></td>
</tr>
<tr>
<td>Patching Product Line Programs [255]</td>
<td></td>
</tr>
<tr>
<td>Early detection of type errors in C++ templates [256]</td>
<td></td>
</tr>
<tr>
<td>Abstracting the Template Instantiation Relation in C++ [257]</td>
<td></td>
</tr>
<tr>
<td>Supporting compile-time debugging and precise error reporting in meta-programs [258]</td>
<td></td>
</tr>
<tr>
<td>Debugging Wireless Sensor Networks with Incremental Snapshots [259]</td>
<td></td>
</tr>
<tr>
<td>An integrated implementation framework for compile-time metaprogramming [260]</td>
<td></td>
</tr>
<tr>
<td>Understanding and Maintaining C++ Generic Libraries [261]</td>
<td></td>
</tr>
<tr>
<td>C-strider: Type-Aware Heap Traversal for C [246]</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.1: Independent citations of my research.
| Thesis 1: Analysis of unity build |  |
| Thesis 2: TMP instrumentation | • • • • • • |
| Thesis 3: Methods for TMP debugging | • • • • |
| Thesis 4: Methods for TMP profiling | • • • • • |
| Thesis 5: Type-preserving heap profiler | |
| title | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 10 |
| Generatív programok helyessége [262] | • | • | • |
| Template metaprogramok hatékony fejlesztése [16] | • | • | • | • | • |
| Unit testing of C++ template metaprograms [263] | • | • |
| Domain-specific language integration with compile-time parser generator library [264] | • |
| Functional Extensions to the Boost Metaprogram Library [265] | • |
| C++ Metastring Library and Its Applications [266] | • |
| Functional Programming with C++ Template Metaprograms [44] | • |
| Syntax check of embedded SQL in C++ with Proto [267] | • |
| Testing by C++ template metaprograms [268] | • • | • |
| Compile-Time Advances of the C++ Standard Template Library [269] | • |

Table 8.2: Fellow citations of my research.
<table>
<thead>
<tr>
<th>description</th>
<th>relevant theses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>application</strong></td>
<td></td>
</tr>
<tr>
<td>Templight: A C++ Template Metaprogram Debugger and Profiler [14, 185]</td>
<td>1</td>
</tr>
<tr>
<td>Template Instantiation Observer in clang [186]</td>
<td>4</td>
</tr>
<tr>
<td>Templight 2.0 - Template Instantiation Profiler and Debugger [184]</td>
<td></td>
</tr>
<tr>
<td><strong>reference</strong></td>
<td></td>
</tr>
<tr>
<td>GoingNative 2013 Q&amp;A Panel [170, 13]</td>
<td>2</td>
</tr>
<tr>
<td>How to find frequently instantiated templates in C++? [270] question on Stack Overflow</td>
<td></td>
</tr>
<tr>
<td>Support for debugging template instantiations [271] KDevelop feature request</td>
<td></td>
</tr>
<tr>
<td>Técnicas de Profiling [272]</td>
<td></td>
</tr>
<tr>
<td>MoDCS Research Group Workshop</td>
<td></td>
</tr>
<tr>
<td>Profiling template metaprogram compilation time [273] question on Stack Overflow</td>
<td></td>
</tr>
<tr>
<td>[MSVC] Meta/template profiler [204]</td>
<td></td>
</tr>
<tr>
<td>Visual C++ feature request</td>
<td></td>
</tr>
<tr>
<td>C++ Object Type Identifier for Heap Profiling (“Type Profiler” in Chromium) [245]</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Applications of and references to my research.
8.1 Results

Thesis 1 (Analysis of unity build). I analysed the unity build technique and measured its effectiveness to reduce build times focusing primarily on full build times. Based on case studies with three open source libraries I determined the pros and cons of applying unity build on existing projects and derived recommended approaches for its implementation. I defined a set of criteria that allow automatic detection of unwanted silent semantic changes caused by the unification. I also derived an algorithm for the implementation of this equivalence check.

Thesis 2 (Template metaprogram instrumentation). I developed a method for instrumenting C++ template metaprograms. The basic concept of the method is the pattern based recognition of template constructs and the insertion of warning generator code fragments. Based on the method I implemented a prototype framework that outputs a position-correct trace file containing all necessary information to reconstruct the full TMP execution. The framework can be adapted to new compilation environments easily.

Thesis 3 (Methods for template metaprogram debugging). I developed two methods for implementing debuggers for C++ template metaprograms. Both method is based on a trace file describing the template events. Besides the non-intrusive data extraction method that utilises the instrumentation technique of Thesis 2, I presented one that is based on compiler modification, and has some advantages over the non-intrusive version. I showed three applications that demonstrate how my methods can be used to implement template debugger functionalities. An implementation based on my research is becoming de facto standard for template metaprogram debugging [14, 180, 184, 185, 186].

Thesis 4 (Methods for template metaprogram profiling). I identified one existing and three new methods for implementing profilers for C++ template metaprograms with different strengths. Supported by detailed measurements I pointed out important aspects of applying these methods in practice. An implementation based on my research is becoming de facto standard for template metaprogram profiling [14, 184, 185, 186].

Thesis 5 (Type-preserving heap profiler). I developed a method for preserving type information in C++ heap profilers. The method is platform independent, and has low memory footprint. I provided optimal algorithms for the performance critical timeline view visualisation. I demonstrated the applicability of the approach in real projects by pinpointing and fixing a performance bug in an open source text editor.
CHAPTER 8. SUMMARY

<table>
<thead>
<tr>
<th>thesis name</th>
<th>relevant publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis of unity build</td>
<td>•</td>
</tr>
<tr>
<td>Template metaprogram instrumentation</td>
<td>• • • • • • •</td>
</tr>
<tr>
<td>Methods for template metaprogram debugging</td>
<td>• • • • • •</td>
</tr>
<tr>
<td>Methods for template metaprogram profiling</td>
<td>• • • • • • •</td>
</tr>
<tr>
<td>Type-preserving heap profiler</td>
<td>•</td>
</tr>
</tbody>
</table>

8.2 Future plans

The next major area of large scale development efficiency that I address is product line variability, especially when it comes down to compile time code variability. In many systems the code base is highly configurable through parameters at preprocessor level [275]. We can find a gap very similar to TMP diagnostic here. The commonly used quality related tools and processes applied today typically simplify the situation by treating different configurations of the same product as different products from their viewpoint. TMP is an exotic usage of generic language constructs in C++ that does not fit well to the standard workflows that try to cover many languages. Similarly compile time variability, which is available only in preprocessed languages, has special needs to be addressed by the tools that want to be as language independent as possible. My current goal is to provide methods for effectively analysing compile time variability of large C++ systems.
References


REFERENCES


154


REFERENCES


REFERENCES


REFERENCES


165
REFERENCES


REFERENCES


REFERENCES


171


Appendices
Appendix A

Generative metaprogramming

A.1 Preprocessor metaprogramming

```c
#include <stdio.h>
#include "matrix.h"

int main()
{
    MATRIX_float mf;
    mf.mRows = (float**)malloc(2 * sizeof(float*));
    mf.mRows[0] = (float*)malloc(2 * sizeof(float));
    mf.mRows[1] = (float*)malloc(2 * sizeof(float));

    MATRIX_double md;
    md.mRows = (double**)malloc(2 * sizeof(double*));
    md.mRows[0] = (double*)malloc(2 * sizeof(double));
    md.mRows[1] = (double*)malloc(2 * sizeof(double));

    mf.mRows[0][1] = 42.0;
    md.mRows[1][0] = 43.0;

    printf("%g/%g\n", MATRIX_float_Get(&mf, 0, 1),
               MATRIX_double_Get(&md, 1, 0));
}
```

Listing A.1: Using the generix matrix implementation (see Figure A.1)
A.1. PREPROCESSOR METAPROGRAMMING

Figure A.1: Generic matrix implementation in C. This is a simplified version of the generic matrix implementation of OpenSceneGraph [54].
APPENDIX A. GENERATIVE METAPROGRAMMING

Listing A.2: Example application of the X macro and the direct product patterns.

```c
#define BUILTIN_COLORS \
X(red, 0xff, 0x00, 0x00) \ 
X(green, 0x00, 0xff, 0x00) \ 
X(blue, 0x00, 0x00, 0xff) \ 
X(black, 0x00, 0x00, 0x00) \ 
X(white, 0xff, 0xff, 0xff)

#define RGB_VALUE(r, g, b) (((r)<<16)|((g)<<8)|(b))

void PrintBuiltinColors()
{
    // Print the names and RGB values of the predefined colors.
    BUILTIN_COLORS
    #undef X
}

void GetRGBByName(const char* name, int* pr, int* pg, int* pb)
{
    // Get the RGB values for a given color name.
    BUILTIN_COLORS
    #undef X
}

const char* GetNameByRGB(int r, int g, int b)
{
    // Return the name of the color by its RGB values.
    const char* colorname;
    switch(RGB_VALUE(r, g, b))
    {
    #define X(colorname, r, g, b) \
    case RGB_VALUE(r, g, b):
        #undef X
        return #colorname;
    #undef BUILTIN_COLORS
    default:
        return "[unknown]";
    } #undef X
}
```

176
# include <boost/preprocessor.hpp>

#define MTX2ARRAY_OP(r, data, i, elem) 
BOOST_PP_COMMA_IF(i) {BOOST_PP_SEQ_ENUM(elem)}
#define MTX2ARRAY(M, array) 
static const size_t array##_r = BOOST_PP_SEQ_SIZE(M); 
static const size_t array##_c = 
BOOST_PP_SEQ_SIZE(BOOST_PP_SEQ_ELEM(0,M)); 
const int array[array##_r][array##_c] = { 
BOOST_PP_SEQ_FOR_EACH_I(MTX2ARRAY_OP, 
BOOST_PP_SEQ_SIZE(M),M) 
};

#define MTX_MUL_ELEM_OP(z, k, M1M2ij) 
(BOOST_PP_MUL(BOOST_PP_SEQ_ELEM(k, 
BOOST_PP_SEQ_ELEM(BOOST_PP_SEQ_ELEM(2,M1M2ij), 
BOOST_PP_SEQ_ELEM(0,M1M2ij)))), 
BOOST_PP_SEQ_ELEM(BOOST_PP_SEQ_ELEM(3,M1M2ij), 
BOOST_PP_SEQ_ELEM(k, BOOST_PP_SEQ_ELEM(1,M1M2ij))))
#define MTX_MUL_CELL_ADD_OP(s,partialsum,elem) 
BOOST_PP_ADD(partialsum,elem)
#define MTX_MUL_CELL_OP(z, j, M1M2i) 
(BOOST_PP_SEQ_FOLD_LEFT(MTX_MUL_CELL_ADD_OP, 0, 
BOOST_PP_REPEAT_FROM_TO(0, 
BOOST_PP_SEQ_SIZE(BOOST_PP_SEQ_ELEM(1,M1M2i)), 
MTX_MUL_ELEM_OP, M1M2i(j))))
#define MTX_MUL_ROW_OP(z, i, M1M2) 
(BOOST_PP_REPEAT_FROM_TO(0, 
BOOST_PP_SEQ_SIZE(BOOST_PP_SEQ_ELEM(0, 
BOOST_PP_SEQ_ELEM(1,M1M2)))), MTX_MUL_CELL_OP, M1M2(i))
#define MTX_MUL(M1, M2) 
BOOST_PP_REPEAT_FROM_TO(0, BOOST_PP_SEQ_SIZE(M1), 
MTX_MUL_ROW_OP, (M1)(M2))

Listing A.3: Using the Boost.Preprocessor framework to implement compile

time matrix multiplication.
APPENDIX A. GENERATIVE METAPROGRAMMING

```
#define MTX1 ((1)(2)(3)) ((3)(4)(5))
#define MTX2 ((1)(2)) ((3)(4)) ((5)(6))

MTX2ARRAY(MTX1, m1)
MTX2ARRAY(MTX2, m2)
MTX2ARRAY(MTX_MUL(MTX1, MTX2), m1mulm2)
```

Listing A.4: Using preprocessor time matrix multiplication (see Listing A.3). See Listing A.5 for the corresponding preprocessor output.

```
static const size_t m1_r = 2;
static const size_t m1_c = 3;
const int m1[m1_r][m1_c] = {
    {1, 2, 3} ,
    {3, 4, 5} },

static const size_t m2_r = 3;
static const size_t m2_c = 2;
const int m2[m2_r][m2_c] = {
    {1, 2} ,
    {3, 4} ,
    {5, 6} };
static const size_t m1mulm2_r = 2;
static const size_t m1mulm2_c = 2;
const int m1mulm2[m1mulm2_r][m1mulm2_c] = {
    {22, 28} ,
    {40, 52} };
```

Listing A.5: Preprocessor output of the matrix multiplication usage example in Listing A.4. Trailing backslashes with accompanying newlines are not part of the original output, I added them for better readability.
A.2 Template metaprogramming

```cpp
#include <type_traits>

struct NullType {};

template<class HeadType = NullType, class TailType = NullType>
struct TypeList
{
    typedef HeadType Head;
    typedef TailType Tail;
};

template<class TypeList, int Index>
struct TypeAt
{
    typedef typename TypeAt<typename TypeList::Tail, Index - 1>::type type;
};

template<class TypeList>
struct TypeAt<TypeList, 0>
{
    typedef typename TypeList::Head type;
};

typedef TypeList<short, TypeList<int, TypeList<long>>> Integers;

typedef typename TypeAt<Integers, 2>::type IntegerTypeAtIndex2;

// long

static_assert(std::is_same<IntegerTypeAtIndex2, long>::value, "error");
```

Listing A.6: Type list with an indexing metafunction.
template <int v0, int v1, int v2, int v3, int v4>
struct Vector5 {
    static const int value0 = v0;
    static const int value1 = v1;
    static const int value2 = v2;
    static const int value3 = v3;
    static const int value4 = v4;
};

template<class, int>
struct Vector5At;
template<class Vec>
struct Vector5At<Vec, 0> {
    static const int value = Vec::value0;
};
... 
template<class Vec>
struct Vector5At<Vec, 4> {
    static const int value = Vec::value4;
};

typedef Vector5<2,3,5,7,11> First5Primes;
static const int ThirdPrime = Vector5At<First5Primes,2>::value; // 5

Listing A.7: Template class with 5 integer parameters is actually a fixed 5 length compile time array of integers.

int f(int a, int b);
int g(int a, int b, int c);

bind(f, 5, _1)(x); // f(5, x)
bind(f, _2, _1)(x, y); // f(y, x)
bind(g, _1, 9, _1)(x); // g(x, 9, x)
bind(g, _3, _3, _3)(x, y, z); // g(z, z, z)
bind(g, _1, _1, _1)(x, y, z); // g(x, x, x)
bind(f, bind(g, _1))(x); // f(g(x))

std::remove_if( first, last, !bind( &X::visible, _1 ) ); // remove invisible objects
std::find_if( first, last, bind( &X::name, _1 ) == "Peter" ||
    bind( &X::name, _1 ) == "Paul" );
std::sort( first, last, bind( &X::name, _1 ) < bind( &X::name, _2 ) ); // sort by name

Listing A.8: Usage examples of Boost.Bind from [276].
A.2. TEMPLATE METAPROGRAMMING

```cpp
#include <iostream>

template<class T>
struct InstanceCounter
{
    InstanceCounter() { ++InstanceCount; }
    ~InstanceCounter() { --InstanceCount; }

    static int InstanceCount;
};
template<class T>
int InstanceCounter<T>::InstanceCount = 0;

struct A : public InstanceCounter<A>
{
};

struct B : public InstanceCounter<B>
{
};

int main()
{
    A a1, a2, a3;
    B b1, b2;
    std::cout << "number of A objects = " << A::InstanceCount << std::endl;
    std::cout << "number of B objects = " << B::InstanceCount << std::endl;
    return 0;
}
```

Listing A.9: CRTP pattern used for implementing instance counting as a mixin [277].
Appendix B

Template metaprogram instrumentation

B.1 Recognition patterns in Templight

functionBeginToken

\[
\text{anytoken} - (\text{';'} | '{'} | {'})') \]

classToken

\[
\text{anytoken} - (\text{';'} | '{'} | \text{class} | \text{struct}) \]

templateFunctionBegin

\[
\text{template} \rightarrow \text{functionBeginToken} \rightarrow \}
\]

Figure B.1: Templight grammar to detect template constructs.
B.1. RECOGNITION PATTERNS IN TEMPLIGHT

Figure B.1: (continued) Templight grammar to detect template constructs.
APPENDIX B. TEMPLATE METAPROGRAM INSTRUMENTATION

B.2 The Spirit grammar used in Templight

```cpp
root = boost::spirit::CLASSIC
    eps_p[boost::bind(&ParseObserver<IteratorType>::StartOfInput, self.GetObserver(), _1, _2)] >>
    *(openBrace[boost::bind(&ParseObserver<IteratorType>::OpenBrace , self.GetObserver() , _1, _2)] | closeBrace[boost::bind(&ParseObserver<IteratorType>::CloseBrace , self.GetObserver() , _1, _2)] | templateFunctionDeclaration | templateFunctionDefinition[boost::bind(&ParseObserver<IteratorType>::TemplateFunction, self.GetObserver() , _1, _2)] | templateClassDeclaration | templateClassDefinition[boost::bind(&ParseObserver<IteratorType>::TemplateClass, self.GetObserver() , _1, _2)] | typedefDeclaration[boost::bind(&ParseObserver<IteratorType>::Typedef, self.GetObserver(), _1, _2, boost::bind(&Grammar::GetLastTypedefIdentifierAndKeywordPosition, &self))] | endOfInput[boost::bind(&ParseObserver<IteratorType>::EndOfInput, self.GetObserver() , _1, _2)] | boost::spirit::CLASSIC anychar_p);
```

```cpp
endOfInput = token_lit (boost::wave::T_EOF) |
    token_lit (boost::wave::T_EOL);
openBrace = token_lit (boost::wave::T_LEFTBRACE);
closeBrace = token_lit (boost::wave::T_RIGHTBRACE);
templateClassBegin = token_lit (boost::wave::T_TEMPLATE) >>
    *(boost::spirit::CLASSIC anychar_p - (token_lit (boost::wave::T_SEMICOLON) |
    token_lit (boost::wave::T_LEFTBRACE) |
    token_lit (boost::wave::T_CLASS) |
    token_lit (boost::wave::T_STRUCT)) >>
    +((token_lit (boost::wave::T_CLASS) | token_lit (boost::wave::T_STRUCT)) >>
    *(boost::spirit::CLASSIC anychar_p - (token_lit (boost::wave::T_CLASS) |
    token_lit (boost::wave::T_CLASS))) |
```
B.2. THE SPIRIT GRAMMAR USED IN TEMPLIGHT

templateClassDeclaration = templateClassBegin >> token_lit (boost::wave::T_SEMICOLON);
templateClassDefinition = templateClassBegin >> token_lit (boost::wave::T_LEFTBRACE);

templateFunctionBegin = token_lit (boost::wave::T_TEMPLATE) >>
+(*(boost::spirit::CLASSIC anychar_p - (token_lit (boost::wave::T_SEMICOLON) |
  token_lit (boost::wave::T_LEFTBRACE) |
  token_lit (boost::wave::T_RIGHTPAREN)) >>
  token_lit (boost::wave::T_RIGHTPAREN)));
templateFunctionDeclaration = templateFunctionBegin >>
token_lit (boost::wave::T_SEMICOLON);
templateFunctionDefinition = templateFunctionBegin >>
token_lit (boost::wave::T_LEFTBRACE);

typedefDeclaration = (token_lit (boost::wave::T_TYPEDEF) >>
boost::spirit::CLASSIC eps_p)[boost::bind &Grammar::RememberTypedefKeywordPosition,
const_cast<Grammar*>(&self), _1, _2] >>
+(*(boost::spirit::CLASSIC anychar_p - (token_lit (boost::wave::T_SEMICOLON) |
  token_lit (boost::wave::T_LEFTBRACE)) |
  token_lit (boost::wave::T_RIGHTPAREN)) >>
  token_lit (boost::wave::T_RIGHTPAREN)) >>
  token_lit (boost::wave::T_IDENTIFIER) |
  token_lit (boost::wave::T_IDENTIFIER) >>
  +(*(token_lit (boost::wave::T_IDENTIFIER) >>
    +(*(token_lit (boost::wave::T_IDENTIFIER) >>
      +(*(token_lit (boost::wave::T_IDENTIFIER) >>
        boost::spirit::CLASSIC anychar_p - (token_lit (boost::wave::T_SEMICOLON) |
        token_lit (boost::wave::T_LEFTBRACE)) | token_lit (boost::wave::T_IDENTIFIER)) >>
        token_lit (boost::wave::T_IDENTIFIER) |
        token_lit (boost::wave::T_IDENTIFIER) >>
        boost::spirit::CLASSIC eps_p) [boost::bind &Grammar::RememberTypedefIdentifier,
const_cast<Grammar*>(&self), _1, _2]) >>
  token_lit (boost::wave::T_SEMICOLON);
APPENDIX B. TEMPLATE METAPROGRAM INSTRUMENTATION

B.3 A complete Templight workflow example

![Diagram of Templight workflow]

Figure B.2: Templight control flow with the listings of the Fibonacci example.

186
B.3. A COMPLETE TEMPLIGHT WORKFLOW EXAMPLE

```cpp
#include <iostream>

// Fibonacci template
template <int N>
struct Fibonacci {
    enum { value = Fibonacci<N-1>::value + Fibonacci<N-2>::value };
};

// Base cases
template <>
struct Fibonacci<0> {
    enum { value = 0 };
};

template <>
struct Fibonacci<1> {
    enum { value = 1 };
};

// Main function
int main() {
    return Fibonacci<ARGUMENT>::value;
}
```

Listing B.2: Original source code of the Fibonacci example.
APPENDIX B. TEMPLATE METAPROGRAM INSTRUMENTATION

Listing B.3: Preprocessed source code of the Fibonacci example with \#line directives.

```cpp
#line 22 "fibonacci.cpp"

int main()
{
    return Fibonacci<5>::value;
}
```

Listing B.4: File/line mapping of the Fibonacci example.

```
1:#line 1 "fibonacci.cpp"
22:#line 22 "fibonacci.cpp"
```

Listing B.5: Unaltered preprocessed source of the Fibonacci example without \#line directives.

```cpp
template <int N>
struct Fibonacci
{
    enum { value = Fibonacci<N-1>::value + Fibonacci<N-2>::value
};

template <>
struct Fibonacci<0>
{
    enum { value = 0
};

template <>
struct Fibonacci<1>
{
    enum { value = 1
};

int main()
{
    return Fibonacci<5>::value;
}
```

188
Listing B.6: Annotation of the Fibonacci example.
Listing B.7: Line mapping of the Fibonacci example.

```cpp
namespace Templight { template<class> struct ReportTemplateBegin { static const unsigned Value = -1.0; }; }
namespace Templight { template<class> struct ReportTemplateEnd { static const unsigned Value = -1.0; }; }
namespace Templight { template<class, class Type> struct ReportTypedef { static const unsigned Value = -1.0; }; }
namespace Templight { template<class, class Type> struct ReportTypedefEvaluation { typedef Type Result; static const unsigned Value = -1.0; }; }

template<int>
struct _T_E_M_P_L_I_G_H_T_functions {
  static void f() {}
};
template <int N>
struct Fibonacci {
  enum _T_E_M_P_L_I_G_H_T___0f { _T_E_M_P_L_I_G_H_T___0value };
  static const unsigned _T_E_M_P_L_I_G_H_T___0 = Templight::ReportTemplateBegin<_T_E_M_P_L_I_G_H_T___0f>::Value;
  enum { value = Fibonacci<N-1>::value + Fibonacci<N-2>::value };
  enum _T_E_M_P_L_I_G_H_T___1f { _T_E_M_P_L_I_G_H_T___1value };
  static const unsigned _T_E_M_P_L_I_G_H_T___1 = Templight::ReportTemplateEnd<_T_E_M_P_L_I_G_H_T___1f>::Value;};
template <>
struct Fibonacci<0> {
  enum _T_E_M_P_L_I_G_H_T___2f { _T_E_M_P_L_I_G_H_T___2value };
  static const unsigned _T_E_M_P_L_I_G_H_T___2 = Templight::ReportTemplateBegin<_T_E_M_P_L_I_G_H_T___2f>::Value;
  enum { value = 0 };
  enum _T_E_M_P_L_I_G_H_T___3f { _T_E_M_P_L_I_G_H_T___3value };
  static const unsigned _T_E_M_P_L_I_G_H_T___3 =
```
B.3. A COMPLETE TEMPLIGHT WORKFLOW EXAMPLE

Listing B.8: Instrumented code of the Fibonacci example.

```cpp
int main()
{
  return Fibonacci<5>::value;
}
```

fibonacci.cpp.patched.cpp

1 fibonacci.cpp.patched.cpp
2 fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing': conversion from 'double' to 'const unsigned int', possible loss of data
3 fibonacci.cpp.patched.cpp(24) : see reference to class template instantiation 'Templight::ReportTemplateBegin<<unnamed-symbol>>' being compiled with
4 [ 
5   "unnamed-symbol"=Fibonacci<0>::_T_E_M_P_L_I_G_H_T___2f
6 ]
7 fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing': truncation of constant value
8 fibonacci.cpp.patched.cpp(2) : warning C4244: 'initializing': conversion from 'double' to 'const unsigned int', possible loss of data
9 fibonacci.cpp.patched.cpp(28) : see reference to class template instantiation 'Templight::ReportTemplateEnd<<unnamed-symbol>>' being compiled with
10 [ 
11 ]
12 191
APPENDIX B. TEMPLATE METAPROGRAM INSTRUMENTATION

```cpp
<unnamed-symbol>=Fibonacci<0>::
  _T_E_M_P_L_I_G_H_T___3f
]
```

```
fibonacci.cpp.patched.cpp(2) : warning C4309: 'initializing':
  truncation of constant value
fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing':
  conversion from 'double' to 'const unsigned int', possible
  loss of data
fibonacci.cpp.patched.cpp(34) : see reference to class
  template instantiation
    'Templight::ReportTemplateBegin<<unnamed-symbol>>'
  being compiled
with
[
<unnamed-symbol>=Fibonacci<1>::
  _T_E_M_P_L_I_G_H_T___4f
]
fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing':
  truncation of constant value
fibonacci.cpp.patched.cpp(2) : warning C4244: 'initializing':
  conversion from 'double' to 'const unsigned int', possible
  loss of data
fibonacci.cpp.patched.cpp(38) : see reference to class
  template instantiation
    'Templight::ReportTemplateEnd<<unnamed-symbol>>'
  being compiled
with
[
<unnamed-symbol>=Fibonacci<1>::
  _T_E_M_P_L_I_G_H_T___5f
]
fibonacci.cpp.patched.cpp(2) : warning C4309: 'initializing':
  truncation of constant value
fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing':
  conversion from 'double' to 'const unsigned int', possible
  loss of data
fibonacci.cpp.patched.cpp(14) : see reference to class
  template instantiation
    'Templight::ReportTemplateBegin<<unnamed-symbol>>'
  being compiled
with
[
<unnamed-symbol>=Fibonacci<5>::
  _T_E_M_P_L_I_G_H_T___0f
]
fibonacci.cpp.patched.cpp(45) : see reference to class
  template instantiation 'Fibonacci<N>' being compiled
with
```
B.3. A COMPLETE TEMPLIGHT WORKFLOW EXAMPLE

[177x770]

fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing': truncation of constant value
fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing': conversion from 'double' to 'const unsigned int', possible loss of data

fibonacci.cpp.patched.cpp(14) : see reference to class template instantiation 'Templight::ReportTemplateBegin<<unnamed-symbol>>' being compiled
with
[131x681]
<unnamed-symbol>=Fibonacci<4>::TEMPLIGHT___0f
fibonacci.cpp.patched.cpp(15) : see reference to class template instantiation 'Fibonacci<N>' being compiled
with
[131x657]
N=4
fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing': truncation of constant value
fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing': conversion from 'double' to 'const unsigned int', possible loss of data

fibonacci.cpp.patched.cpp(14) : see reference to class template instantiation 'Templight::ReportTemplateBegin<<unnamed-symbol>>' being compiled
with
[129x633]
<unnamed-symbol>=Fibonacci<3>::TEMPLIGHT___0f
fibonacci.cpp.patched.cpp(15) : see reference to class template instantiation 'Fibonacci<N>' being compiled
with
[129x617]
N=3
fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing': truncation of constant value
fibonacci.cpp.patched.cpp(1) : warning C4244: 'initializing': conversion from 'double' to 'const unsigned int', possible loss of data
APPENDIX B. TEMPLATE METAPROGRAM INSTRUMENTATION

loss of data

fibonacci.cpp.patched.cpp(14) : see reference to class
template instantiation
'Templight::ReportTemplateBegin<<unnamed-symbol>>'
being compiled

with
|
70 <unnamed-symbol>=Fibonacci<2>::
_T_E_M_P_L_I_G_H_T___0f
|
72 fibonacci.cpp.patched.cpp(15) : see reference to class
template instantiation 'Fibonacci<N>' being
compiled
|
73 with
|
74 |
75 N=2
|
76 }
|
77 fibonacci.cpp.patched.cpp(1) : warning C4309: 'initializing'
truncation of constant value
|
78 fibonacci.cpp.patched.cpp(2) : warning C4244: 'initializing'
conversion from 'double' to 'const unsigned int', possible
loss of data
|
79 fibonacci.cpp.patched.cpp(18) : see reference to class
template instantiation
'Templight::ReportTemplateEnd<<unnamed-symbol>>'
being compiled
|
80 with
|
81 |
82 <unnamed-symbol>=Fibonacci<2>::
_T_E_M_P_L_I_G_H_T___1f
|
83 }
|
84 fibonacci.cpp.patched.cpp(2) : warning C4309: 'initializing'
truncation of constant value
|
85 fibonacci.cpp.patched.cpp(2) : warning C4244: 'initializing'
conversion from 'double' to 'const unsigned int', possible
loss of data
|
86 fibonacci.cpp.patched.cpp(18) : see reference to class
template instantiation
'Templight::ReportTemplateEnd<<unnamed-symbol>>'
being compiled
|
87 with
|
88 |
89 <unnamed-symbol>=Fibonacci<3>::
_T_E_M_P_L_I_G_H_T___1f
|
90 }
|
91 fibonacci.cpp.patched.cpp(2) : warning C4309: 'initializing'
truncation of constant value
|
92 fibonacci.cpp.patched.cpp(2) : warning C4244: 'initializing'
conversion from 'double' to 'const unsigned int', possible
B.3. A COMPLETE TEMPLIGHT WORKFLOW EXAMPLE

Listing B.9: Compilation messages of the Fibonacci example.

```xml
<?xml version="1.0" standalone="yes"?>
<!-- [trace] generated by Templight -->
<Trace>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|24|1"/>
<Context context = "Fibonacci<0/>"
</TemplateBegin>
<TemplateEnd>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|34|1"/>
<Context context = "Fibonacci<1/>"
</TemplateBegin>
<TemplateEnd>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|48|1"/>
<Context context = "Fibonacci<2/>"
</TemplateBegin>
<TemplateEnd>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|62|1"/>
<Context context = "Fibonacci<3/>"
</TemplateBegin>
<TemplateEnd>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|76|1"/>
<Context context = "Fibonacci<4/>"
</TemplateBegin>
<TemplateEnd>
<TemplateBegin>
<Position position = "fibonacci.cpp.patched.cpp|90|1"/>
<Context context = "Fibonacci<5/>"
</TemplateBegin>
<TemplateEnd>
```
Listing B.10: Trace file of the Fibonacci example before position adjust.
B.3. A COMPLETE TEMPLIGHT WORKFLOW EXAMPLE

Listing B.11: Position correct trace file of the Fibonacci example.
Appendix C

Template metaprogram debugging and profiling

C.1 Trace file with memoisation events

```xml
<?xml version="1.0" standalone="yes"?>
<Trace>
  <TemplateBegin>
    <Kind>TemplateInstantiation</Kind>
    <Context context = "Fibonacci&lt;2&gt;"/>
    <PointOfInstantiation>fibonacci.cpp|25|10
      <TimeStamp time = "468767508.019546"/>
      <MemoryUsage bytes = "570976"/>
    </PointOfInstantiation>
  </TemplateBegin>
  <TemplateEnd>
    <Kind>TemplateInstantiation</Kind>
    <TimeStamp time = "468767508.025486"/>
    <MemoryUsage bytes = "571584"/>
  </TemplateEnd>
  <TemplateBegin>
    <Kind>Memoization</Kind>
    <Context context = "Fibonacci&lt;2&gt;"/>
    <PointOfInstantiation>fibonacci.cpp|25|10
      <TimeStamp time = "468767508.025558"/>
      <MemoryUsage bytes = "571552"/>
    </PointOfInstantiation>
  </TemplateBegin>
  <TemplateEnd>
    <Kind>Memoization</Kind>
    <TimeStamp time = "468767508.025579"/>
    <MemoryUsage bytes = "571552"/>
  </TemplateEnd>
</Trace>
```
Listing C.1: Trace file with profile information and memoisation events generated by a modified version of the clang compiler [14]. The input source code was the Fibonacci example in Listing B.2 with \texttt{ARGUMENT=2}.
Appendix D

Type-preserving heap profiler for C++

D.1 Real usage statistics of timeline view

<table>
<thead>
<tr>
<th>$d$ (s)</th>
<th>$\delta$ (ms)</th>
<th>$\omega$</th>
<th>$n_e$</th>
<th>$n_a$</th>
<th>$n_d$</th>
<th>$n_d/n_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>1</td>
<td>33150</td>
<td>201</td>
<td>1382901</td>
<td>25545929</td>
<td>18</td>
</tr>
<tr>
<td>251</td>
<td>50</td>
<td>5019</td>
<td>401</td>
<td>3664114</td>
<td>158501462</td>
<td>43</td>
</tr>
<tr>
<td>251</td>
<td>10</td>
<td>25089</td>
<td>701</td>
<td>3664114</td>
<td>272057852</td>
<td>74</td>
</tr>
<tr>
<td>18855</td>
<td>10000</td>
<td>1886</td>
<td>487</td>
<td>272360</td>
<td>213899354</td>
<td>785</td>
</tr>
<tr>
<td>51330</td>
<td>10</td>
<td>5132957</td>
<td>351</td>
<td>695057</td>
<td>4503929</td>
<td>6</td>
</tr>
<tr>
<td>51330</td>
<td>1</td>
<td>51329567</td>
<td>51</td>
<td>695057</td>
<td>343579</td>
<td>0</td>
</tr>
<tr>
<td>51330</td>
<td>1</td>
<td>51329567</td>
<td>351</td>
<td>695057</td>
<td>2775155</td>
<td>4</td>
</tr>
<tr>
<td>166193</td>
<td>10000</td>
<td>16620</td>
<td>421</td>
<td>16062</td>
<td>7166680</td>
<td>446</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>421</td>
<td>225516</td>
<td>137136114</td>
<td>608</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>421</td>
<td>286124</td>
<td>167375947</td>
<td>585</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>421</td>
<td>596777</td>
<td>207833275</td>
<td>348</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>166194</td>
<td>4196</td>
<td>596777</td>
<td>2065440995</td>
<td>3461</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>421</td>
<td>1083018</td>
<td>209664008</td>
<td>194</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>166194</td>
<td>4196</td>
<td>1083018</td>
<td>2079324607</td>
<td>1920</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>166193</td>
<td>942</td>
<td>1083018</td>
<td>955293534</td>
<td>882</td>
</tr>
<tr>
<td>805297</td>
<td>400</td>
<td>2013242</td>
<td>446</td>
<td>143222</td>
<td>39226340</td>
<td>274</td>
</tr>
</tbody>
</table>

Table D.1: Real usage statistics of timeline view, partial views.
### D.1. Real Usage Statistics of Timeline View

<table>
<thead>
<tr>
<th>d (s)</th>
<th>δ (ms)</th>
<th>ω</th>
<th>n_t</th>
<th>n_a</th>
<th>n_d</th>
<th>n_d/n_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
<td>673</td>
<td>674</td>
<td>516902</td>
<td>83023814</td>
<td>161</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>720</td>
<td>721</td>
<td>124723</td>
<td>45732303</td>
<td>367</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>813</td>
<td>814</td>
<td>267921</td>
<td>61416943</td>
<td>229</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>968</td>
<td>969</td>
<td>762615</td>
<td>209182253</td>
<td>274</td>
</tr>
<tr>
<td>24</td>
<td>20</td>
<td>1180</td>
<td>1181</td>
<td>268058</td>
<td>159744208</td>
<td>596</td>
</tr>
<tr>
<td>32</td>
<td>50</td>
<td>625</td>
<td>626</td>
<td>875410</td>
<td>125956159</td>
<td>144</td>
</tr>
<tr>
<td>34</td>
<td>50</td>
<td>664</td>
<td>665</td>
<td>1382901</td>
<td>83864354</td>
<td>61</td>
</tr>
<tr>
<td>34</td>
<td>10</td>
<td>3316</td>
<td>3317</td>
<td>1382901</td>
<td>413742915</td>
<td>299</td>
</tr>
<tr>
<td>37</td>
<td>100</td>
<td>365</td>
<td>366</td>
<td>2462</td>
<td>522138</td>
<td>212</td>
</tr>
<tr>
<td>37</td>
<td>100</td>
<td>368</td>
<td>369</td>
<td>271212</td>
<td>49425760</td>
<td>182</td>
</tr>
<tr>
<td>48</td>
<td>50</td>
<td>952</td>
<td>953</td>
<td>1841069</td>
<td>252708953</td>
<td>137</td>
</tr>
<tr>
<td>49</td>
<td>100</td>
<td>491</td>
<td>492</td>
<td>1612823</td>
<td>159900539</td>
<td>99</td>
</tr>
<tr>
<td>49</td>
<td>20</td>
<td>2448</td>
<td>2449</td>
<td>1612823</td>
<td>793281658</td>
<td>492</td>
</tr>
<tr>
<td>53</td>
<td>1000</td>
<td>53</td>
<td>54</td>
<td>262513</td>
<td>283659</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>100</td>
<td>521</td>
<td>522</td>
<td>262513</td>
<td>471453</td>
<td>2</td>
</tr>
<tr>
<td>71</td>
<td>100</td>
<td>701</td>
<td>702</td>
<td>1200632</td>
<td>134327629</td>
<td>112</td>
</tr>
<tr>
<td>74</td>
<td>100</td>
<td>738</td>
<td>739</td>
<td>2348835</td>
<td>262269853</td>
<td>112</td>
</tr>
<tr>
<td>105</td>
<td>200</td>
<td>522</td>
<td>523</td>
<td>137063</td>
<td>57957291</td>
<td>423</td>
</tr>
<tr>
<td>107</td>
<td>1000</td>
<td>107</td>
<td>108</td>
<td>78216</td>
<td>1319754</td>
<td>17</td>
</tr>
<tr>
<td>107</td>
<td>100</td>
<td>1063</td>
<td>1064</td>
<td>78216</td>
<td>12397039</td>
<td>158</td>
</tr>
<tr>
<td>212</td>
<td>200</td>
<td>1059</td>
<td>1060</td>
<td>200939</td>
<td>171394997</td>
<td>853</td>
</tr>
<tr>
<td>251</td>
<td>500</td>
<td>503</td>
<td>504</td>
<td>3664114</td>
<td>108101573</td>
<td>30</td>
</tr>
<tr>
<td>320</td>
<td>1000</td>
<td>320</td>
<td>321</td>
<td>146119</td>
<td>42224498</td>
<td>289</td>
</tr>
<tr>
<td>320</td>
<td>1000</td>
<td>320</td>
<td>321</td>
<td>267053</td>
<td>84290267</td>
<td>316</td>
</tr>
<tr>
<td>320</td>
<td>500</td>
<td>640</td>
<td>641</td>
<td>5013961</td>
<td>170256273</td>
<td>34</td>
</tr>
<tr>
<td>320</td>
<td>200</td>
<td>1599</td>
<td>1600</td>
<td>5013961</td>
<td>418050101</td>
<td>83</td>
</tr>
<tr>
<td>320</td>
<td>100</td>
<td>3198</td>
<td>3199</td>
<td>5013961</td>
<td>831078165</td>
<td>166</td>
</tr>
<tr>
<td>331</td>
<td>1000</td>
<td>331</td>
<td>332</td>
<td>275962</td>
<td>78884642</td>
<td>286</td>
</tr>
<tr>
<td>331</td>
<td>1000</td>
<td>331</td>
<td>332</td>
<td>592153</td>
<td>157580941</td>
<td>266</td>
</tr>
<tr>
<td>331</td>
<td>500</td>
<td>662</td>
<td>663</td>
<td>4409870</td>
<td>351849222</td>
<td>80</td>
</tr>
<tr>
<td>331</td>
<td>200</td>
<td>1652</td>
<td>1653</td>
<td>4409870</td>
<td>846030158</td>
<td>192</td>
</tr>
<tr>
<td>331</td>
<td>100</td>
<td>3303</td>
<td>3304</td>
<td>4409870</td>
<td>1741480988</td>
<td>395</td>
</tr>
</tbody>
</table>

Table D.2: Real usage statistics of timeline view, full views, d < 1000.
<table>
<thead>
<tr>
<th>(d) (s)</th>
<th>(\delta) (ms)</th>
<th>(\omega)</th>
<th>(n_t)</th>
<th>(n_a)</th>
<th>(n_d)</th>
<th>(n_d/n_a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1232</td>
<td>1000</td>
<td>1233</td>
<td>1234</td>
<td>1257</td>
<td>1476276</td>
<td>1174</td>
</tr>
<tr>
<td>2075</td>
<td>4000</td>
<td>519</td>
<td>520</td>
<td>93081</td>
<td>31301117</td>
<td>336</td>
</tr>
<tr>
<td>2451</td>
<td>1000</td>
<td>2452</td>
<td>2453</td>
<td>7349</td>
<td>13850945</td>
<td>1885</td>
</tr>
<tr>
<td>3614</td>
<td>4000</td>
<td>905</td>
<td>906</td>
<td>100167</td>
<td>73234142</td>
<td>731</td>
</tr>
<tr>
<td>16994</td>
<td>1000</td>
<td>16994</td>
<td>16995</td>
<td>1584</td>
<td>25983504</td>
<td>16404</td>
</tr>
<tr>
<td>18855</td>
<td>10000</td>
<td>1886</td>
<td>1887</td>
<td>272360</td>
<td>400438407</td>
<td>1470</td>
</tr>
<tr>
<td>28700</td>
<td>1000</td>
<td>28701</td>
<td>28702</td>
<td>1713</td>
<td>47360811</td>
<td>27648</td>
</tr>
<tr>
<td>45708</td>
<td>100000</td>
<td>458</td>
<td>459</td>
<td>631344</td>
<td>2740459</td>
<td>4</td>
</tr>
<tr>
<td>51330</td>
<td>100000</td>
<td>514</td>
<td>515</td>
<td>695057</td>
<td>3227684</td>
<td>5</td>
</tr>
<tr>
<td>51330</td>
<td>10000</td>
<td>5134</td>
<td>5135</td>
<td>695057</td>
<td>25980003</td>
<td>37</td>
</tr>
<tr>
<td>51330</td>
<td>1000</td>
<td>51331</td>
<td>51332</td>
<td>695057</td>
<td>253467542</td>
<td>365</td>
</tr>
<tr>
<td>61625</td>
<td>100000</td>
<td>617</td>
<td>618</td>
<td>722415</td>
<td>1520359</td>
<td>2</td>
</tr>
<tr>
<td>61625</td>
<td>1000</td>
<td>6163</td>
<td>6164</td>
<td>722415</td>
<td>8687055</td>
<td>12</td>
</tr>
<tr>
<td>61625</td>
<td>10000</td>
<td>61625</td>
<td>61626</td>
<td>722415</td>
<td>80359508</td>
<td>111</td>
</tr>
<tr>
<td>166193</td>
<td>10000</td>
<td>16620</td>
<td>16621</td>
<td>16062</td>
<td>92438204</td>
<td>5755</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>16621</td>
<td>225516</td>
<td>2409446815</td>
<td>10684</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>16621</td>
<td>286124</td>
<td>2904118748</td>
<td>10150</td>
</tr>
<tr>
<td>166194</td>
<td>100000</td>
<td>167</td>
<td>168</td>
<td>596777</td>
<td>33917156</td>
<td>57</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>1662</td>
<td>1663</td>
<td>596777</td>
<td>327863586</td>
<td>549</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>16621</td>
<td>596777</td>
<td>327934737</td>
<td>5481</td>
</tr>
<tr>
<td>166194</td>
<td>100000</td>
<td>1662</td>
<td>1663</td>
<td>1083018</td>
<td>329235649</td>
<td>304</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>16620</td>
<td>16621</td>
<td>1083018</td>
<td>3280423740</td>
<td>3029</td>
</tr>
<tr>
<td>166194</td>
<td>10000</td>
<td>166194</td>
<td>166195</td>
<td>1083018</td>
<td>3278882727</td>
<td>30275</td>
</tr>
<tr>
<td>805297</td>
<td>200000</td>
<td>403</td>
<td>404</td>
<td>1432222</td>
<td>57857266</td>
<td>404</td>
</tr>
<tr>
<td>805297</td>
<td>200000</td>
<td>4027</td>
<td>4028</td>
<td>1432222</td>
<td>57664805</td>
<td>4026</td>
</tr>
<tr>
<td>805297</td>
<td>20000</td>
<td>40265</td>
<td>40266</td>
<td>1432222</td>
<td>5764364445</td>
<td>40248</td>
</tr>
<tr>
<td>805297</td>
<td>20000</td>
<td>402649</td>
<td>402650</td>
<td>1432222</td>
<td>57641694335</td>
<td>402464</td>
</tr>
</tbody>
</table>

Table D.3: Real usage statistics of timeline view, full views, \(d \geq 1000\).
Appendix E

Dissertation summaries
APPENDIX E. DISSERTATION SUMMARIES

E.1 Dissertation summary

Developers face different challenges when working with large C++ systems: slow builds, comprehension difficulties, compile and run time efficiency. These challenges are more intense when generative metaprogramming techniques are used. Full compilation time can be substantially reduced with unity build. In Thesis 1 I analysed unity build and measured its effectiveness. Based on case studies I determined the pros and cons of applying unity build on existing projects and derived recommended approaches for its implementation. I defined a set of criteria that allow automatic detection of unwanted silent semantic changes caused by the unification, and derived an algorithm for the implementation of this equivalence check.

While diagnostic functionalities are natural parts of most programming environments, this was not true for C++ template metaprogramming. I set the foundations and developed methods for template metaprogram (TMP) instrumentation (Thesis 2), debugging (Thesis 3) and profiling (Thesis 4). My instrumentation method is applicable even if the source code of the compiler or the necessary knowledge for its modification is not available. For debugging and profiling I presented more portable non-intrusive and more powerful compiler modifying approaches. For profiling I identified one existing and three new methods, where my claims about overhead are supported by measurements.

For effective memory behaviour analysis, a type-aware heap profiler is a must. Contrary to other OO languages, C++ does not have direct support for tracking types of allocated objects. In Thesis 5 I presented a platform independent, low footprint method for preserving type information in a C++ heap profiler. I provided optimal algorithms for the performance critical timeline view visualisation. I demonstrated the applicability of the approach in real projects by pinpointing and fixing a performance bug in an open source text editor.

With Theses 1 and 5 I elevated the former tricks to become readily applicable techniques even in industrial environments. With Theses 2, 3 and 4 I supplied working solutions to the obvious, but unsatisfied needs. An implementation based on my research is becoming de facto standard for template metaprogram debugging and profiling. The numerous citations to my first research paper on TMP diagnostics indicates it was indeed a pioneer work.
E.2. DISSZERTÁCIÓ ÖSSZEFOGLALÓ

E.2. Disszertáció összefoglaló

Nagyméretű C++ rendszerekkel történő munka során a fejlesztők különböző kihívásokkal szembesülnek: lassú fordítás, a kód megértésének nehézsége, fordítási és futási időbeli hatékonyság. Ezek a kihívások generatív metaprogramozási technikák használata esetén intenzívebbek.

A teljes fordítási idő jelentősen csökkenthető egyesített fordítással. Az 1-es Tézisben elemeztem az egyesített fordítást és megmértem a hatékonyságát. Esetnélmények alapján meghatároztam az egyesített fordítás létező projektekre történő alkalmazásának előnyeit és hátrányait, és az implementáció javasolt megközelítését vezettem le. Definiáltam egy kritériumhalmazt, mely lehetővé teszi az egyesítés okozta nemkívánt, észrevétlen szemantikus változások automatikus detektálását, és ezen azonosságenerősítés implementációjához levezettem egy algoritmust.

Míg a diagnosztikai funkciók a legtöbb programozási környezet természetes részei, ez nem volt igaz a C++ metaprogramozásra. Megteremtettem ezek alapjait és módszereket fejlesztettem ki template metaprogram (TMP) instrumentálásra (2-es Tézis), nyomkövetésre és hibakeresésre (3-as Tézis), valamint profilozásra (4-es Tézis). Az instrumentáló módszerem akkor is alkalmazható, ha a fordító forráskódja, vagy a módosításához szükséges tudás nem áll rendelkezésre. Nyomkövetésre és profilozásra hordozhatóbb nem-intruzív és teljesebb, de a fordító módosításával járó megközelítéseket mutattam be. Profilozásra egy létező és három új módszert azonosítottam, melyeknél a terheléstöbbletre vonatkozó állításaimat mérések támasztják alá.

A memóriahasználat hatékon elemzéséhez egy típusos heap profilozó elengedhetetlen. Más OO nyelvekkel ellentétben a C++ közvetlenül nem támogatja a lefoglalt objektumok típusának követését. Az 5-ös Tézisben egy platformfüggetlen, kis terheléstöbbletű módszert mutattam be típusinformációk megőrzésére egy C++ heap profilozóban. Optimális algoritmusokat nyújtottam a teljesítménymértékek idővonal nézet megjelenítésére. Egy nyílt forrásszöveg szerkesztő teljesítményhibájának beazonosításával és javításával szemléltettem a megközelítés valós projektekben történő alkalmazhatóságát.

Az 1-es és 5-ös Tézissel az azelőtti „trükköket” akár ipari környezetben is készen alkalmazható technikákká emeltem. A 2-es, 3-as és 4-es Tézissel működő megoldásokat adtam a nyilvánvaló, de kielégítetlen igényekre. A kutatásomra alapul megvalósítás a template metaprogram nyomkövetés és profilozás de facto szabványának válós, A TMP diagnosztikáról szóló első tudományos cikkem számos idezése mutatja, hogy valóban úttörő munka volt.