Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 97-115

ANALYSIS OF PROFILING TECHNIQUES FOR C++ TEMPLATE METAPROGRAMS

Z. Porkoláb, J. Mihalicza, N. Pataki and Á. Sipos

(Budapest, Hungary)

Abstract. Template metaprogramming (TMP) is an emerging new direction in C++ programming for executing algorithms in compilation time. Despite all of its already proven benefits and numerous successful applications, TMP is yet to become an accepted technique in industrial projects. One reason is the lack of professional software tools supporting the development of template metaprograms. On the other hand, a strong analogue between traditional runtime programs and compile-time metaprograms presents the possibility for creating development tools similar to those already used when writing runtime programs. This paper presents two methods for metaprogram profiling. Firstly, *Templight*, a debugging and profiling framework is introduced. The framework reveals the steps executed by the compiler during the compilation of C++ programs with templates. Thus Templight is capable of adding timestamps to template instantiations and measuring their times. The second method uses compiler modification acquiring instantiation profiling data from the compiler itself.

1. Introduction

Code efficiency is an all-important aspect of software design. In order to improve the efficiency of a program, a programmer must identify the critical parts. As static analysis methods in many cases fail to explore the dynamical behavior of the program, execution profiling is a key element to finding bottlenecks in the code. In order to investigate the behavior of a program profilers should collect

information during the runtime. Boehm reports that 20 percent of the routines consumes 80 percent of the execution time [4]. Knuth claimes that less than 4 percent of a program usually accounts for more than 50 percent of its runtime [7].

The more complex the language environment we work in, the more sophisticated profiling toolset we need. In object-oriented languages like C++ tools must be able to measure all elements of classes, like constructors and destructors, inlined functions, static variables etc. [12]. Most current profilers [29, 30] available for object-oriented languages are capable of handling these problems.

The everyday process of programming consists of compiling, executing and profiling the code. However, the recently emerged programming paradigm, *C++ template metaprogramming* (TMP) does not follow this pattern. In template metaprogramming the program itself is running during compilation time. A cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [21]. The output of this process is still checked by the compiler and runs as an ordinary program. TMP is based on the C++ *templates*. Templates are key language elements for the C++ programming language [15, 18] and are essential for capturing commonalities of abstractions. *Generic programming* [14] is a recently emerged programming paradigm for writing highly reusable components. The *Standard Template Library (STL)* – the most notable example of generic programming – is now an unavoidable part of most professional C++ programs [8].

Template metaprograming is proved to be a Turing-complete sublanguage of C++ [5]. We write metaprograms for various reasons, here we list some of them:

- *Expression templates* [22] replace runtime computations with compile-time activities to enhance runtime performance.
- *Static interface checking* increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [9, 13].
- *Active libraries* [19]. Active libraries act dynamically during compile-time, making decisions based on programming contexts and making optimizations.

Unfortunately, implementations of template metaprograms are typically far from optimal [1]. One reason is that compilers are optimized to generate efficient runtime code and not designed to maximize efficiency of the compilation process itself. Another reason is that programmers are not familiar with all the background costs of the metaprogram constructs. This may result in a very long compilation time and huge memory usage. With a profiling tool we should be able to identify these ”noisy” code segments, which hold up the compilation process. Since traditional profiler tools are unapplicable to metaprograms running

in compile-time, the development of metaprogram-specific profiling tools is crucial. Unfortunately, today there are no TMP profiling tools available. In this paper we propose two methods for TMP profiling, which serve as foundations of an optimization process. Our first method is instrumenting the C++ source to emit well-formed messages during compilation. Information gained this way during the running of a metaprogram is used to measure the compilation process. Other methods modify the compiler itself producing profiling information on C++ template instantiations and memory usage.

This paper is organized as follows. In Section 2 we give an overview of C++ template metaprogramming compared to runtime programming. Section 3 describes profiling in general. Our external profiling framework is described in details in Section 4. Our second approach, presented in Section 5, involves the modification of the open source `g++` compiler. In Section 6 we analyze our results. Limitations and future directions are discussed in Section 7. In Section 8 we overview the most important related works.

2. C++ template metaprograms

In our context the notion *template metaprogram* stands for the collection of templates, their instantiations and specializations, whose purpose is to carry out operations in compile-time. Their expected behavior might be either emitting messages or generating special constructs for the runtime execution. Henceforth we will call a *runtime program* any kind of runnable code, including those which are the results of template metaprograms.

Conditional statements (and stopping recursion) are solved via specializations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [5].

Data is expressed in runtime programs as constant values or literals. In metaprograms we use `static const` and enumeration values to store quantitative information. Results of computations during the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, the execution of a metaprogram may trigger the creation of new types by the compiler. These types may hold information that influences the further execution of the metaprogram.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree

structures or sequences. Tree structures are the favorite implementation forms of expression templates [22]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [6].

However, there is a fundamental difference between runtime programs and C++ template metaprograms: once a certain entity (constant, enumeration value, type) has been defined, it will be immutable. There is no way to change its value or meaning. A metaprogram does not contain assignments. In this sense metaprogramming is similar to pure functional programming languages, where *referential transparency* is obtained. That is the reason why we use recursion and specialization to implement loops: we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect, too: unwanted side effects do not occur.

3. Profilers

Profilers are software tools carrying out performance analysis by measuring the runtime behavior of programs. The most commonly analyzed behaviors are the frequency and duration of subprogram calls and the used heap memory's size. These *events* are either recorded into a *trace*, a stream of recorded events, or a *profile*, a statistical summary of the observed events. Profilers use numerous techniques to measure softwares, including hardware interrupts, operating system hooks, performance counters and *code instrumentation*.

Instrumentation is a process, during which the profiler modifies the analyzed program, inserting profiling code fragments. Instrumentations can be executed manually by the programmer or automatically by the compiler. Instrumentation may be a binary translation when the tool adds instrumentation to a compiled binary code. Another method is runtime instrumentation, when the code is instrumented directly before the execution. In this case the analyzed software is controlled by the profiler. The profiler may work with runtime injection, i.e. the code is modified at runtime.

Another profiling method is sampling. These profilers are called statitistical profilers. A sampling profiler probes the analyzed software's program counter at regular intervals using operating system interrupts. Sampling profilers are typically less accurate and specific, but allow the measured software to run at near full speed.

Since instrumentations are not the part of the analyzed code, profiling has overhead [26]. Instrumentations are typically added at specific points to the analyzed software's code. These points are called *instrumentation points* (IP). An instrumentation point encapsulates the functionality of instrumentation and the IP's original program context. An instrumentation point consists of an instru-

mentation probe and instrumentation payload. The *payload* is the activity that collects the data about the measured program. The *probe* is the activity that switches the analyzed code to the payload. The probe may have a condition that controls the invocation of the payload.

The number of executed probes, or the probe count causes instrumentation overhead. The probe count for an instrumentation point is the number of the instances of probe. All executed probes incur overhead associated with executing the probe code that intercepts program execution. Because of the condition that controls the payload, every instance of a probe may not incur overhead of the payload. Thus, the total overhead for an instrumentation point is related to the number of done probes, how much each probe costs, how frequently the payload is called and the payload's cost.

Instrumentation is a widely used technique in profilers, but this method incurs overhead. Our *Templight* framework also uses instrumentation for profiling. In Section 6.1 the overhead of Templight is examined, its performance is compared to the modified compiler.

4. Modification of the source code

Most compilers generate additional information for profilers. An appropriate compiler support for measuring template metaprogram profiles would be the ideal solution. However, as this support is unavailable as of now, an immediate and portable method is to use external tools cooperating with standard C++ language elements.

Without the modification of the compiler the only way of obtaining any information about our metaprogram during compilation is to generate warning messages [1]. Therefore the task is the *instrumentation* of the source, i.e. its transformation into a functionally equivalent modified form that triggers the compiler to emit talkative warning messages. The inserted code fragments are designed to generate warnings that contain enough information about the context and details of the actual event. Whenever the compiler instantiates a template, defines an inner type etc. the inserted code fragments generate detailed information on the actual template-related event. A pipeline transmits the information for the profiler which measures the time of the event. As we will see in the results, for large template metaprograms the overhead of the communication between processes is negligible.

The instrumented code fragments, on the other hand, result in big performance overhead, that can significantly distort the measured data.

The instrumentation is based on the *Templight* framework, designed mainly

for debugging C++ template metaprograms [10]. The framework was intentionally designed to be as portable as possible, for this end we tried to use portable and standard-compliant tools. Almost all components are written in standard C++ using the `STL`, `boost` and `Xerces` libraries.

The input of Templight is a C++ source file and the output is a trace file, a list of events like *instantiation of template X began*, *instantiation of template X ended*, *typedef definition found* etc.

The procedure begins with the execution of the preprocessor, followed by invoking the `boost::wave` C++ parser. Our aim is to insert warning-generating code fragments at the instrumentation points. As `wave` does not do semantic analysis we can only recognise these places by searching for specific token patterns. We go through the token sequence and look for patterns like *template keyword + arbitrary tokens + class or struct keyword + arbitrary tokens +* { to identify template definitions. This pattern matching step is called annotating, its output is an XML file containing annotation entries in a hierarchical structure following the scope.

The instrumentation takes this annotation and the single source and inserts the warning-generating code fragments for each annotation at its corresponding location in the source, thus producing a source that emits warnings at each annotation point during its compilation. The next step is the execution of the compiler to have these warning messages generated. The inserted code fragments are intentionally designed to generate warnings that contain enough information about the context and details of the actual event. Since the compiler may produce output independently of our instrumentation, it is important for debugger warnings to have a distinct format that differentiates them. This is the step where we ask the compiler for valuable information from its internals. Here the result is simply the build output as a text file. The warning translator takes the built output, looks for the warnings with the aforementioned special format and generates an event sequence with all the details. The result is an XML file that lists the events that occurred during the compilation in chronological order. For profiling purposes, timestamps are also placed in the XML file for each instantiation. Following is a segment of the file with profiling data:

```
<TemplateBegin>
    <Position position = "prof1.cpp.patched.cpp|12|1"/>
    <Context context = "fibonacci&lt;127&gt;"/>
    <History>
        <TemplateContext
            instance="Templight::ReportTemplateBegin&lt;
                C,__formal&gt;">
            <Parameter name="C" value="fibonacci&lt;
                127&gt;::_T_E_M_P_L_I_G_H_T___0f"/>
```

```
            <Parameter name="__formal" value="pointer-to-member(0x0)"/>
        </TemplateContext>
        <TemplateContext
            instance="fibonacci&lt;n&gt;">
            <Parameter name="n" value="127"/>
        </TemplateContext>
    </History>
    <TimeStamp time = "9F8F2950-01C67659"/>
</TemplateBegin>
```

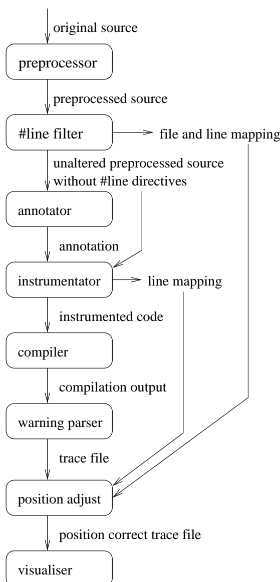

Figure 1. Architecture of debugging/profiling framework

Compile-time performance is discussed in [1] with a spectacular test. Since there was no other applicable tool, the authors had to fall back on measuring the full compilation time and modifying a preprocessor parameter every time thus producing measurement series and graphs. With the Templight framework we have to execute only one compilation that emits warnings for each instantiation, and a post processing pipelined tool memorizes the timestamps whenever a warning occures. This way we have timestamps for each template-related event, and the processing time of a certain template instance can be easily computed by subtracting the timestamps stored at the corresponding template-begin and template-end event (warning message).

5. Modification of the compiler

Our next method for profiling compilation times of template metaprograms is compiler-dependent.

The most accurate way for evaluating compilers is by acquiring timing information from the compiler itself. As our metaprogram is executed on a meta-level from the viewpoint of C++, a meta-level profiler is needed, i.e. one measuring the compiler's action times. The obvious approach is to use a profiler tool (like `gprof`) and measure the compiler's runtime. Even though we would be able to measure `instantiate_class_template`'s running time in general, we could not disambiguate certain instantiations. In other words, we could acquire the sum of all instaniation times, but would not be able to measure each instantiation separately. So this approach does not fulfil our requirements.

To gain the required detailed data on particular instantiations we have to modify the compiler. To demonstrate this method we have chosen the widely used GNU `g++` compiler, as its C source code is freely available, thus rendering it a plausible target for "hacking", and developing possible future compiler features. In the center or our examination is the `instantiate_class_template` function. In `g++`, instantiation of a new type is done in this function, which resides in the `pt.c` file. This function does not handle full specializations (which are important for ending metaprogram recursions), and does not look up already instantiated types, this is done in other parts of `g++`. On the other hand we do not need timing data of these compilation operations either, as we only want to measure instantiaton times.

Some of the desired information could be obtained by executing the `g++` back-end, `cc1plus`. This is a process doing the actual compiling, and when called directly prints debug information, like parsing times, name lookup times and others. Again, the problem with these timing data is that they refer to the whole compilation process and not distinct instantiations.

The next step is the modification of the compiler. Our test-phase profiler for `g++` is implemented as follows: we have modified our `g++` 3.4.0 to record timestamps when template instantiations begin and end, thus the time differences mean the time it takes to instantiate one particular template. Two calls to `gettimeofday` are placed to the beginning and very end of mentioned function. Note that even though this function has more return commands, which obviously end the function call (and the instantiation), these deal with cases like erroneous syntax. We do not have to take these into consideration when analyzing a sound metaprogram [10]. The results of the measurements can be recorded in more ways, resulting in different sets of data. These can be used for a comparative study of analysis methods.

We implemented the compiler modifications in two variants. The first method prints the data to the screen at the moment the data is obtained, at the bottom of the compiler's instantiation function. This turned out to be a non-authentic way of acquiring profiling information, as printing results in considerable time loss due to the time it takes to carry out I/O operations. Not only does this `printf` slow down the compilation (in our test with Example 1 around 20.000 lines are printed) it seriously distorts profiling data. In fact many operating system-dependent properties might affect profiling data, like the size of the screen, the size of the screen buffer, etc.

The second version stores timing data in arrays, whose matter is printed at the end of the compilation. We conjectured this approach would lead to more precise data than that of the first method, because of the I/O operations, buffering, redirections, etc. However, the first method produced similar results when we chose to redirect all printing to the standard output into a file (see Section 6).

6. Evaluation of the methods

In this section we present our measurement results with the proposed methods. We analyzed the profiling methods from the following points of view:

1. *Accuracy.* What is the accuracy of the different profiler methods?
2. *Applicability.* Are the profilers applicable to large programs?
3. *Overhead.* What are the overheads of the profiling methods itselves: i.e. to what degree do the different methods distort profiling results?

When constructing the tests we partly followed the examples discussed by Abrahams and Gurtovoy in [1]. In these examples the importance of *memoization* is emphasized. Memoization is a procedure done by the compiler when instantiating new types from templates. Each instantiation begins with a lookup in the compiler's repository, searching for the type about to be instantiated. If the compiler does find the type (i.e. it has already been instantiated) it aborts the creation procedure and uses the already finished type. Memoization speeds up the compilation [1], as this lookup happens much faster than it would take to repeat the instantiation. In order to avoid this phenomenon, we modified our `fibonacci` metaprogram (computing Fibonacci numbers in compile-time) to enforce instantiation.

In the following we describe the four test methods whose results are presented in this section.

The first method follows the test method described by Abrahams and Gurtovoy. Here the program was compiled and the full compilation time was measured with the `UNIX` `time` command. Therefore the compilation times of the templated part and the rest of the program were not separated. The curve representing the result data is labelled *g++ whole*.

The second approach uses the Templight framework to instrument the source code. The instrumentation results in a code that emits a warning message at the begin and end points of each template instantiation. As the warnings appear, the external profiling tool measures the time spent on the instantiation. Thus we are able to separate the compilation of templates from the rest of the activities.

The third and fourth methods are based on the modification of the `g++` compiler. Since there was no significant difference between the results with buffering the output and the immediate printing with redirection to file, we illustrated these results with one curve labelled *g++ mod*.

Figure 2 shows the results of our experiments with the same non-memoizating example.

The most significant experience based on the results is that the characteristics of all the three curves are similar. Templight has constant overhead. The best results have been produced by the built-in solution that indicates the importance of compiler supported template metaprogram debuggers.

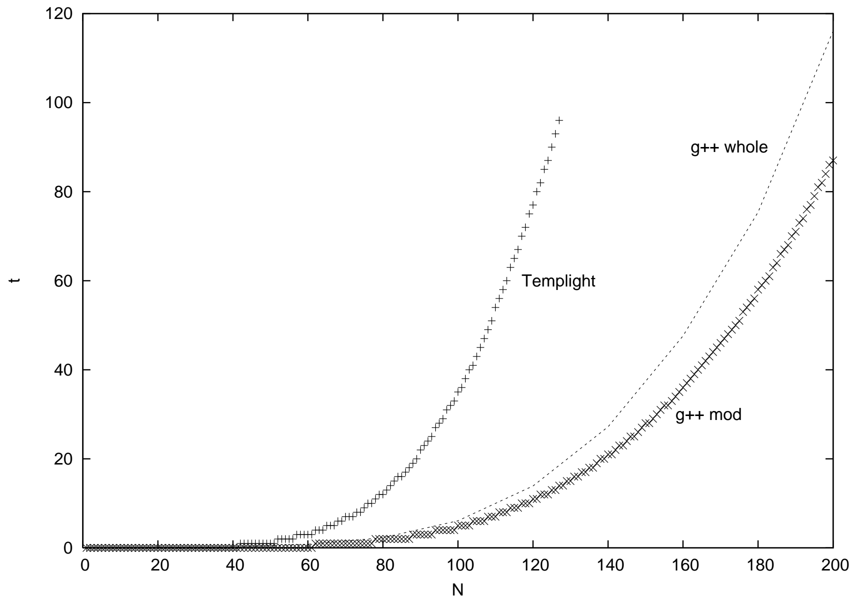

Figure 2. Instantiation time without memoization

6.1. Accuracy

To estimate the accuracy of our methods we compared the total execution time of `g++` to the outermost template's instantiation time, i.e. the running of the whole metaprogram. The difference between `fibonacci<N,N>` ($N = 1, 20, 40...$) and the whole compilation time grew linearly, with $N = 200$ the difference being about 30 seconds. Even though we have acquired the same $O(N^3)$ complexity for the compilation time as found in [1], there is a significant difference between numeric data of the outermost template's instantiation and the whole compilation time.

N	1	20	40	60	80	100	120	140	160	180	200
full	0.1	0.1	0.4	0.8	2.3	6.1	13.9	27.2	47.6	75.4	116.1
inst	0.1	0.1	0.4	0.8	2.3	5.1	11.2	21.3	36.4	58.6	87.8

Table 1. Full compilation time (full) vs. sum time of instantiation (inst)

The cause of the difference is that even in case of high number of template instantiations, all the operations `g++` carries out before, after, and between instantiating types (source code analysis, optimization, code generation, etc.) have heavy costs. The result shows the importance of precise profiling methods.

Template instance	Full instantiation time
fibonacci<118,119>	2,5837152
fibonacci<121,123>	2,6137584
fibonacci<113,120>	2,6237728
fibonacci<124,125>	2,6237728
fibonacci<118,121>	2,6337872
fibonacci<55,55>	2,653816
fibonacci<115,121>	2,6638304
fibonacci<120,122>	2,6638304
fibonacci<123,124>	2,6738448
fibonacci<122,123>	2,703888
fibonacci<117,122>	2,7139024
fibonacci<119,123>	2,75396
fibonacci<121,124>	2,7940176
fibonacci<123,125>	2,8440896
fibonacci<56,56>	2,8440896

Table 2. Instantiation times

6.2. Applicability

One of the most frequently used functionalities of a profiler is the determination of critical parts that slow down the compilation process. Both the modified `g++`, and Templight generate a trace containing the names of the instantiated templates, and the instantiation times. With the help of a script processing this trace file we can easily obtain a list of template instances sorted by their compilation times. A portion of Templight's output profiling `fibonacci` is shown in Table 2.

On the other hand, `fibonacci` is a relatively simple metaprogram. In larger software projects, however, there are many templates referencing each other, and it is not as easy to spot the exact cause of a slow compilation as in the previous example. To see how this profiling technique operates in projects where numerous templates are used we measured the compilation of the Templight framework itself. The measured source combines different template libraries like STL algorithms, `Boost::spirit`, and `Boost::wave`. Table 3 shows the 20 most time-consuming instantiations.

Template instance	Time
boost::iterator_facade<boost::filesystem::bas...	511
boost::detail::iterator_facade_types<const st...	420
boost::detail::facade_iterator_category_impl<...	290
boost::wave::context<char *,Templight::FileAn...	180
Templight::Grammar<boost::wave::pp_iterator<b...	171
boost::wave::util::functor_input::inner<boost...	161
boost::wave::impl::pp_iterator_functor<boost:...	151
boost::wave::util::macromap<boost::wave::cont...	130
boost::spirit::tree_match<boost::wave::cpplex...	91
boost::mpl::if_<boost::detail::is_iterator_ca...	80
boost::detail::operator_brackets_result<boost...	71
boost::mpl::if_<boost::detail::use_operator_b...	71
boost::detail::is_pod_impl<const std::basic_s...	61
boost::detail::is_scalar_impl<const std::basi...	51
boost::mpl::if_<boost::is_convertible<std::bi...	50
boost::mpl::if_<boost::mpl::and_<boost::is_re...	50
boost::spirit::unary<boost::spirit::chlit<cha...	50
boost::spirit::unary<boost::spirit::chlit<wch...	50
boost::spirit::unary<boost::spirit::strlit<co...	50
boost::spirit::tree_node<boost::spirit::node_...	41

Table 3. Compilation times per template instances

If we are not interested in the actual instances, but rather we would like to see what templates need the most time during the compilation process, we can see the table in template level as shown in Table 4. Though none of the instantiations of the STL templates appear in the first table, in the template level view we can see that the `std::allocator` template is instantiated 64 times and takes the 13th most time to be processed.

Template	Time	Count
`boost::iterator_facade`	511	1
`boost::detail::iterator_facade_types`	420	1
`boost::mpl::if_`	411	12
`boost::detail::facade_iterator_category_impl`	290	1
`boost::detail::is_convertible_impl_dispatch_base`	200	8
`boost::call_traits`	190	5
`boost::spirit::unary`	190	4
`boost::wave::util::functor_input::inner`	181	2
`boost::wave::context`	180	1
`Templight::Grammar`	171	1
`boost::wave::impl::pp_iterator_functor`	151	1
`std::allocator`	131	64
`boost::wave::util::macromap`	130	1
`boost::detail::is_abstract_imp`	120	8
`boost::detail::is_pointer_impl`	110	6
`boost::spirit::tree_match`	91	1
`boost::detail::is_convertible_impl`	80	24
`boost::detail::operator_brackets_result`	71	1
`boost::detail::is_pod_impl`	61	5

Table 4. Compilation times per templates

We can easily have a quick overview about the compilation times of the different template library usages in our code if we sum the template processing times by their namespaces. This comparison can be found in Table 5. Tables 4 and 5 not only give an overview of the processing times but also present the number of template instantiations per templates and per namespaces, respectively.

6.3. Overhead

The Templight framework inserts code fragments into the user code, resulting in significant compilation time overhead, see Table 6. Compiler modification methods result only in marginal overheads.

Namespace	Time	Count
boost	4663	819
std	772	241
Templight	181	5

Table 5. Compilation times per namespaces

Test	Overhead
memoisation	+646%
Templight source	+73%

Table 6. Compilation time overhead caused by the inserted code fragments when the Templight framework is used

7. Limitations and future work

7.1. Modification of the source code

Compiler support. The Templight framework works only if the compiler gives enough information when it meets the instrumented erroneous code. Unfortunately not all compilers fulfil this criterion today. Table 7 summarizes our experiences with some compilers.

It is a frequent case when a warning is emitted, but there is no information about its context. The most surprising find was that the Borland 5.6 compiler does not print any warnings to our instrumented statement even with all warnings enabled. A later version of this compiler (version 5.8) prints the desired messages, but similarly to many others it does not generate any context information. In contrast to the others this compiler prints the same warning for each instantiation.

Semantics. Since we do not have semantical information we fall back on using mere syntactic patterns. Unfortunately without semantic information there are ambiguous cases where it is impossible to determine the exact role of the tokens. This simply comes from the environment-dependent nature of the language and from the heavily overloaded symbols. The following line for example can have totally different semantics depending on its environment:

```
enum { a = b < c > :: d };
```

If the preceding line is

compiler	result
g++ 3.3.5	ok
g++ 4.1.0	ok
MSVC 7.1	ok
MSVC 8.0	ok
Intel 9.0	no instantiation backtrace
Comeau 4.3.3	no instantiation backtrace
Metrowerks CodeWarrior 9.0	no instantiation backtrace
Borland 5.6	no warning message at all
Borland 5.8	no instantiation backtrace, but the warning message is printed for each instantiation

Table 7. Our experiences with different compilers

```
enum { b = 1, c = 2, d = 3 };
```

then the `<` and `>` tokens are relational operators, and `::` stands for 'global scope', while having the following part instead of the previous line

```
template<int>
struct b {
    enum { d = 3 };
};
enum { c = 2 };
```

the `<` and `>` tokens become template parameter list parentheses and `::` the dependent name operator. This renders recognising `enum` definitions more difficult.

7.2. Modification of the compiler

Inheritance. Apart from the obvious disadvantage of modifying compiler source code, the most important limitation of our simple g++ modification is the lack of support for inheritance. Consider

```
template <int N>
struct A
{
...
```

```
};
template <int N>
struct Factorial : public A<N>
{
...
};
```

When measuring the instantiation time of `Factorial<N>` with some integer N, only the time of the instantiation of `Factorial<N>`'s body will be considered. Our test cannot take into account the time it takes to instantiate parent classes' bodies. This is a problem when a metaprogram relies heavily on inheritance, like `boost::mpl`, `boost::wave`, etc.

As mentioned in Section 5, full specializations of templates are not taken into consideration in our profiling data sets, since they are concrete types from the start and need no instantiation. On the other hand, a possible future direction would be measuring the lookup times of full specializations, and also already instantiated templates.

8. Related work

Template metaprogramming was first investigated in Veldhuizen's articles [21]. Vandevoorde and Josuttis introduced the concept of a *tracer*, which is a specially designed class that emits runtime messages when its operations are called [18]. When this type is passed to a template as an argument, the messages show in what order and how often the operations of that argument class are called. The authors also defined the notion of an *archetype* for a class whose sole purpose is checking that the template does not set up undesired requirements on its parameters.

To improve the compilation of heavily templated C++ programs Veldhuizen proposed alternative compilation models [20], each with a distinct tradeoff of compile time, code size and code speed.

In their book on `boost` [1] Abrahams and Gurtovoy devoted a whole section to diagnostics, where the authors showed methods for generating textual output in the form of warning messages. They implemented the compile-time equivalent of the aforementioned runtime tracer (`mpl::print`). Compile-time performance was also investigated via a set of carefully selected test cases. The cost of memoization, memoized lookup and instantiation was characterized and compared across various compilers.

9. Conclusion

C++ template metaprogramming is a new, evolving programming paradigm. It extends traditional runtime programming with numerous advantages, like implementing active libraries, optimizing numerical operations and enhancing compile-time checking possibilities. Since metaprograms typically show bad compile-time performance, identifying the bottlenecks is a crucial task.

In this article we evaluated profiling techniques applicable for C++ template metaprograms. Instrumenting the C++ source, collecting and measuring the emitted messages during compilation is a highly portable but limited possibility. Real profiling information can only be obtained with the help of the compiler. To demonstrate this possibility we modified the `g++` compiler to produce profiling information on C++ template metaprograms.

References

[1] **Abrahams D. and Gurtovoy A.,** *C++ template metaprogramming. Concepts, tools, and techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[2] **Alexandrescu A.,** *Modern C++ design: Generic programming and design patterns applied*, Addison-Wesley, 2001.

[3] *ANSI/ISO C++ Committee. Programming languages – C++.* ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] **Boehm B.W.,** Improving software productivity, *IEEE Computer* **20** (9) (1987), 43-57.

[5] **Czarnecki K. and Eisenecker U,W.,** *Generative programming: methods, tools and applications*, Addison-Wesley, 2000.

[6] **Karlsson B.,** *Beyond the C++ standard library. An introduction to Boost*, Addison-Wesley, 2005.

[7] **Knuth D.E.,** An empirical study of FORTRAN programs, *Software - Practice and Experience*, **1** (1971), 105-133.

[8] **Musser D.R. and Stepanov A.A.,** Algorithm-oriented generic libraries, *Software - Practice and Experience*, **27** (7) (1994), 623-642.

[9] **McNamara B. and Smaragdakis Y.,** Static interfaces in C++, *First Workshop on C++ Template Metaprogramming*, 2000.

[10] **Porkoláb Z., Mihalicza J. and Sipos Á.,** Debugging C++ template metaprograms, *Proceedings of GPCE 2006, Portland,* ACM Series (accepted)

[11] **Dos Reis G. and Stroustrup B.,** Specifying C++ concepts, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006, 295-308.

[12] **Shende S., Malony A.D., Cuny J., Lindlan K., Beckman P. and Karmesin S.,** Portable profiling and tracing for parallel ccientific applications using C++, *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998, 134-145.

[13] **Siek J. and Lumsdaine A.,** Concept checking: Binding parametric polymorphism in C++, *First Workshop on C++ Template Metaprogramming*, 2000.

[14] **Siek J.,** *A language for generic programming*, PhD Thesis, Indiana University, 2005.

[15] **Stroustrup B.,** *The C++ programming language special edition*, Addison-Wesley, 2000.

[16] **Stroustrup B.,** *The design and evolution of C++*, Addison-Wesley, 1994.

[17] **Unruh E.,** Prime number computation, *ANSI X3J16-94-0075/ISO WG21-462.*

[18] **Vandevoorde D. and Josuttis N.M.,** *C++ templates: The complete guide*, Addison-Wesley, 2003.

[19] **Veldhuizen T.L. and Gannon D.,** Active libraries: Rethinking the roles of compilers and libraries, *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98)*, SIAM Press, 1998, 21-23.

[20] **Veldhuizen T.,** Five compilation models for C++ templates, *First Workshop on C++ Template Metaprogramming*, 2000.

[21] **Veldhuizen T.,** Using C++ template metaprograms, *C++ Report*, **7** (4) (1995), 36-43.

[22] **Veldhuizen T.,** Expression templates, *C++ Report*, **7** (5) (1995), 26-31.

[23] **Zólyomi I., Porkoláb Z. and Kozsik T.,** An extension to the subtype relationship in C++, *GPCE 2003,* LNCS **2830**, 2003, 209-227.

[24] **Zólyomi I. and Porkoláb Z.,** Towards a template introspection library, LNCS **3286**, 2004, 266-282.

[25] **Czarnecki K., Eisenecker U.W., Glck R., Vandevoorde D. and Veldhuizen T.L.,** *Generative programmind and active libraries*, Springer Verlag, 2000.

[26] **Kumar N., Childers B.R. and Soffa M.L.,** Low overhead program monitoring and profiling, *The 6th ACM SIGPLAN-SIGSOFT Workshop on program analysis for software tools and engineering, 2005,* 28-34.

[27] **Graham S., Kessler P. and McKusick M.,** gprof: A call graph execution profiler, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, MA, June 1982*, ACM, 1982, 120-126.

[28] **Gregor D., Jrvi J., Siek J.G., Dos Reis G., Stroustrup B. and Lumsdaine A.,** Concepts: Linguistic support for generic programming in C++, *Proceedings of the 2006 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, 2006.

[29] `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`

[30] `http://www.cs.uoregon.edu/research/tau/home.php`

Z. Porkoláb, J. Mihalicza, N. Pataki and Á. Sipos
Department of Programming Languages
Eötvös Loránd University
Pázmány Péter sét. 1/C
H-1117 Budapest, Hungary
`gsd@elte.hu, pocok@inf.elte.hu, patakino@elte.hu, shp@elte.hu`