Measuring Compilation Time of C++ Template Metaprograms

Zoltán Porkoláb
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
gsd@elte.hu

József Mihalicza
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
jmihalicza@gmail.com

Norbert Pataki
Eötvös Loránd University,
Dept. of Programming Languages
and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary
patakino@elte.hu

Abstract

Template metaprograms have become an essential part of today's C++ programs. Despite their importance there are surprisingly few tools for creating, using and analysing them efficiently. The radically increased compilation time of programs using template-heavy code is one of the phenomena of which root is a serious challange to trace down. The compilation speed can increase when one starts using even less complex template metaprogram structures. Industrial projects pay a big price for these additional times. Our paper presents intrusive and non-intrusive approaches of getting detailed information on the compilation process of templates and measuring time each specific template instantiations had taken. The results show the pros and cons of the different techniques, comparing them across different compilers. In an example we demonstrate how to identify the bottleneck and how to improve template metaprograms regarding compilation time.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification – C++; D.2.5 [*Testing and Debugging*]: Profiling

General Terms Languages

Keywords C++ template metaprogram, Profiling, Compilation time, Tools

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. 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]. To identify and modify these critical parts may significally improve the efficiency of the whole program. As static analysis methods fail in many cases 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.

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 [30, 31] available for object-oriented languages are capable of handling these problems.

Templates are essential part of the C++ language, by enabling data structures and algorithms to be parameterized by types [15, 18]. This abstraction is frequently needed when using general algorithms like finding an element in a data structure, or defining data types like a list or a matrix of elements of same type. The mechanism behind a matrix containing integer or floating point numbers, or even strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can express this abstraction, thus this *generic* language construct aids code reuse, and the introduction of higher abstraction levels. This method of code reuse often called *parametric polymorphism*, to emphasise that here the variability is supported by compile-time template parameter(s).

The effect of heavy usage of templates in C++ had a significant negative effect on compilation time. As templates have to be presented as source code, they are usually implemented in form of header files. Header files are included into source files as well as other header files in a recursive manner. The parser have to process each of them repetedly in every compilation unit.

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]. The STL as well as the rest of the C++ standard library mostly consist of template classes placed into a number of header files. While the usage of templates provides a great flexibility and reusability for the programmers, compile times grew significantly.

The everyday process of programming consists of compiling, executing (and perhaps 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 run as an ordinary program.

Template metaprograming is proved to be a Turing-complete sublanguage of C++ [5]. We write metaprograms for various reasons, like *expression templates* [22] replacing runtime computations with compile-time activities to enhance runtime performance, *static interface checking*, which 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], acting dynamically during compile-time, making decisions based on programming contexts and making optimizations, and many others.

Unfortunately, execution of template metaprograms are typically far from optimal [1]. One reason is that compilers are optimized to generate efficient runtime code and optimal efficiency of the compilation process of average C++ codes. However, template metaprograms have specialities like recursive instantiations, template specialisations, typelists. These constructs are notorius sources of increased compilation time. Another main reason of writing unefficient template metaprograms 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.

C++0x, the next standard of C++ will introduce a number of new features, programmers long time desire. However, experts agree that an additional ten percent of compile time increase will be expected.

In this situation fast build process is a crucial part of project management. To achive this goal we have to understand the compile time behaviour of our programs, identify critical portions, unnecessarily included headers, avoidable instantiations, and expensive compile time structures.

Traditional (i.e. run-time) profiling tools are inadequate as they are impossible to measure compile time activities. Profiling the compiler does not give any relevant information about the compiled code ifself. We need special methods to measure the compilation process, especially the behaviour of templates. With such *template profiling tools* we should be able to identify compilation bottlenecks, "noisy" code segments, which hold up the compilation process.

In this paper we propose methods for profiling heavily templated C++ codes, especially template metaprograms. First we discuss the method of measuring the compilation of the whole program. To detect critical parts, however, we have to analyse the instantiation process of every template individually. For this purpose we extend a template metaprogram debugging framework [10] with time informations. To make the measure more accurate, we show a number of improvements, some of them includes the modification of the compiler itself to move the timestamp generation closer the event. These methods may serve as foundations of an optimization process.

This paper is organized as follows. In Section 2 we give an overview of C++ template metaprogramming compared to runtime programming specially regarding the profiling requirements. Our external profiling framework is described in details in Section 3.2. Our second approach, presented in Section 3.3 involves the modification of the open source g++ compiler. In Section 4 we analyze our results. Limitations of our methods are overviewed in Section 5. Related works and future directions are discussed in Section 5. The paper is concluded in Section 7.

2. Profiling 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.

C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops.

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].

Profilers are software tools carrying out performance analysis by measuring the behavior of programs. The most commonly analyzed behaviors for run-time programs 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*.

When we are looking for compile time analogy of these terms, we can identify certain similarities between run-time and compile time programs. Template classes play similar role in metaprograms like subprograms – functions or procedures – do in run-time case. They control the chain of instantiation, just like ordinary functions can call each other. They have even parameters: types or constants, to change the actual behaviour when different arguments are applied. The chain of instantiation is a fundamental property of inspecting template metaprograms similarly to the execution stack of their run-time equivalences.

Intrumentation 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.

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 instrumentation 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.

Instrumenting templates requires injecting identifyable code snippets into the original source on well-chosen IP places. As our main goal is to chart template instantiation steps, an ideal place for such injection is the syntactical beginning and end of each templates. Thus when templates are instantiated the IP's may trace the chain of instantiation. As the whole metaprogram is executed by the compiler to detect these events are not trivial.

Another popular profiling method for run-time programs is sampling. These profilers are often 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.

Implementing sampling profilers for template metaprograms are possible but their usufulness is far behind their run-time counterparts. It is possible to interrupt the compilation process, and compilers could be modified to report which part of code is executed at the moment. However, as the compilation process – in most logical cases – a deterministic process statistical methods have not too much value here.

Since instrumentations are not the part of the analyzed code, profiling has overhead [26]. 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.

We can avoid this overhead when measuring full programs, rather then their's structural parts. Such test programs can highligh certain patterns of template metaprograms even they do not revealing internal details.

3. Practical approaches of profiling

3.1 Measuring compilation units

The most available method to measure compile time performance is measuring full compilation of units. Compilation of full source files does not require to modify the code, thus this is a non-intrusive method, and do not add overhead or significant distorsion. Although filter out all perturbations are not easy, most of the operating systems provide us fair tools to measure the experienced real-time, user and system times on the run of a compilation session.

In most cases locating, loading, and parsing header files is a non-trivial effort. To filter out this effect we can run the precompiler in a separate session and measure only further compilation stages. Figure 1 shows that separating precompiler tasks changes the compilation times significantly.

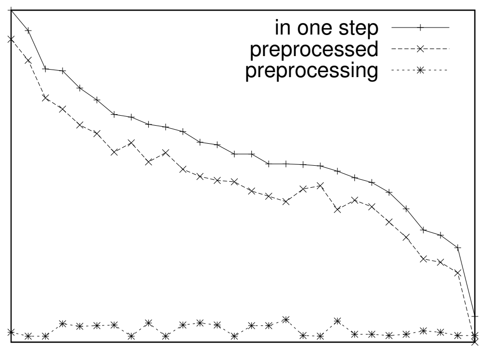

Figure 1. Compilation time with separate precompilation

Compiling full programs or compilation units can reveal significant behavioural patterns of programs or template constructs. Abrahams and Gurtovoy measured template metaprogram constructs in [1] with this method and were able pointing to fundametal differences in strategy and tactics of different compilers. They have shown the effect of certain techniques, like *memoisation* and have measured structural complexity of metaprograms.

However, measuring full compilation time has certain shortages. It is not always trivial to write wrapper programs around certain template constructs without seriously distort measurement results. Full session of compilation includes activities we are not interested in: initializations, outputting, solve non-template related tasks. When we analyse the results we have the compilation times, but no implications on how this gross time splits amongs different code components. Measuring full compilation is great to prove concepts but hard to use for analysis.

3.2 Measuring with instrumenting

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.

Our instrumentation method is based on the *Templight* framework, originally addressed 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 overall architecture of the framework is seen on Figure 2.

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 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

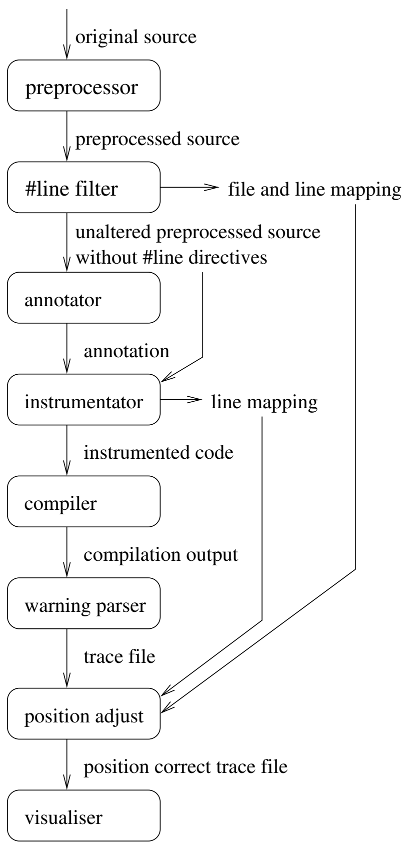

Figure 2. Architecture of debugging/profiling framework

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 build 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:

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).

There is an interesting distortion factor in case of instrumentation regarding inheritance hierarchy. Let consider the following code snippet:

```
template <typename T>
class Derived : public Base<T>
{
  // ..
};
```

When `Derived` is about to be instantiated, the compiler starts to instantiate `Base` with typename parameter `T`. This process will be finished *before* class `Derived` would start to be instantiated. If we inject the observer code at the beginning and end-point of both classes, we end up with the following sequence of emitted warnings:

```
Base starts
Base ends
Derived starts
Derived ends
```

In other words, inheritance relationship would remain hidden, and instantiation time for `Derived` (wrongly) will not include the time of instantiation of `Base`. This is a serious problem, when a metaprogram relies heavily on inheritance, like in the case of `boost::mpl` or `boost::wave::wave` Therefore we have to handle this situation with greater attention.

An other factor of distortion is the way we add timestamps to the emitted warning messages. Compilers do not decorate warnings with timestamp info. In the simplest solution an external program reads compiler output and apply timestamps. In this case the delay between the warning is generated and timestamped can be sigificant. Better way, if timestamp is generated inside the compiler when constructing the warning message, this delay can be eliminated. But this requires the modification of the compiler.

3.3 Modification of the compiler

The most accurate way for evaluating compilation times 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 – to use a profiler tool (like `gprof`) and measure the compiler's runtime – does not work, since we cannot identify those parts of the subject code wich are under compilation. Even though we would be able to measure some kind of compiler method `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.

To gain the required detailed data on particular instantiations we have to modify the compiler fur the purpose. We instrument the code with templight, but generate warnings with timestamp via the modified compiler. To demonstrate this method we have chosen the widely used GNU g++ compiler (version 3.4.3), 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 of our examination is the modification of warning generation procedure, i.e. the `warning` and `cp_warning_at` functions.

The modification consists of generating timestamps when entering and exiting these functions and adding it to the emitted message. We used this approach to eliminate the distortion of generating the warning itself. Experiments showed, however, that most cases the time we spent in these functions is negligible.

4. 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 four 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 3 shows the results of our experiments with the same non-memoizating example.

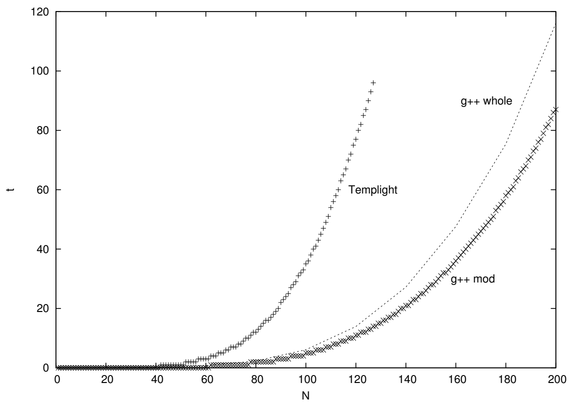

Figure 3. Instantiation time without memoization

The most significate 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.

4.1 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
full	0.1	0.1	0.4	0.8	2.3	6.1	13.9	27.2	47.6	75.4
inst	0.1	0.1	0.4	0.8	2.3	5.1	11.2	21.3	36.4	58.6

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.

4.2 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.

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

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
`iterator_facade<boost::filesystem::bas...`	511
`detail::iterator_facade_types<const st...`	420
`detail::facade_iterator_category_impl<...`	290
`wave::context<char *,Templight::FileAn...`	180
`Templight::Grammar<wave::pp_iterator<b...`	171
`wave::util::functor_input::inner<boost...`	161
`wave::impl::pp_iterator_functor<boost:...`	151
`wave::util::macromap<boost::wave::cont...`	130
`spirit::tree_match<boost::wave::cpplex...`	91
`mpl::if_<boost::detail::is_iterator_ca...`	80
`detail::operator_brackets_result<boost...`	71
`mpl::if_<boost::detail::use_operator_b...`	71
`detail::is_pod_impl<const std::basic_s...`	61
`detail::is_scalar_impl<const std::basi...`	51
`mpl::if_<boost::is_convertible<std::bi...`	50
`mpl::if_<boost::mpl::and_<boost::is_re...`	50
`spirit::unary<boost::spirit::chlit<cha...`	50
`spirit::unary<boost::spirit::chlit<wch...`	50
`spirit::unary<boost::spirit::strlit<co...`	50
`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
`iterator_facade`	511	1
`detail::iterator_facade_types`	420	1
`mpl::if_`	411	12
`detail::facade_iterator_category_impl`	290	1
`detail::is_convertible_impl_dispatch_base`	200	8
`call_traits`	190	5
`spirit::unary`	190	4
`wave::util::functor_input::inner`	181	2
`wave::context`	180	1
`Templight::Grammar`	171	1
`wave::impl::pp_iterator_functor`	151	1
`std::allocator`	131	64
`wave::util::macromap`	130	1
`detail::is_abstract_imp`	120	8
`detail::is_pointer_impl`	110	6
`spirit::tree_match`	91	1
`detail::is_convertible_impl`	80	24
`detail::operator_brackets_result`	71	1
`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. Table 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.

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

Table 5. Compilation times per namespaces

4.3 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.

Test	Overhead
memoisation	+646%
Templight source	+73%

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

5. Limitations

5.1 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.

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

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

```
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.

6. Related work and future directions

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.

Abrahams and Coelho compared compilers and compilation strategies in [29]. They pointed to such important parameters, like reinstantiation overhead, and symbol size.

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.

Accuracy is a fundamental property of profilers. To minimize overhead of instrumentation and measurement we have to step forward to the modification of compilers. While instumentation is a more portable solution, real industrial solutions we could imagine via compiler modifications only.

In this paper we have concentrated mostly on temporal properties of metaprogram execution. Memory consuption is at least as important as the run-time properties. We want to extend our researches to this area.

Using relaible template metaprogram profilers it is possible to describe the compile time characteristics of metaprogram algorithms. In STL, runtime complexity guarantees are part of the standard. Similar compile time guaranties for template metaprograms can largely improve practicability of template metaprograms.

In C++0x concepts will support constrained generics, i.e. the programmer may define *concepts* – requirements against template arguments. While this will be a long time desired functionality for C++ community, most notably programmers for the STL library, early tests warn that the new feature may increase compilation time. One of our future research plans involve the examination of this language feature, especially the compile-time consequences the heavy usage of concept maps, and searching for best practices.

7. 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] David Abrahams, Aleksey Gurtovoy: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston, 2004.

[2] Andrei Alexandrescu: 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] Barry W. Boehm: Improving Software Productivity. IEEE Computer 20, Vol. 9, 1987, pp. 43-57.

[5] Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)

[6] Björn Karlsson: Beyond the C++ Standard Library, A Introduction to Boost. Addison-Wesley, 2005.

[7] Donald E. Knuth: An Empirical Study of FORTRAN Programs. Software - Practice and Experience 1, 1971, pp. 105-133.

[8] David R. Musser and Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.

[9] Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[10] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos: Debugging C++ Template Metaprograms. Accepted for publication in the ACM series, in the proceedings of GPCE 2006, October, 2006, Portland.

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

[12] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin: Portable Profiling and Tracing for Parallel Scientific Applications using C++. In Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, August 1998, pp. 134-145.

[13] Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[14] Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.

[15] Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)

[16] Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)

[17] Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.

[18] David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)

[19] Todd L. Veldhuizen and Dennis Gannon: Active libraries: Rethinking

the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23

[20] Todd Veldhuizen: Five compilation models for C++ templates. In First Workshop on C++ Template Metaprogramming, October 2000

[21] Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[22] Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[23] István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.

[24] István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.

[25] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glck, David Vandevoorde, Todd L. Veldhuizen: Generative Programmind and Active Libraries. Springer-Verlag, 2000

[26] Naveen Kumar, Bruce R. Childers, Mary Lou Soffa: Low overhead program monitoring and profiling. The 6th ACM SIGPLAN-SIGSOFT Workshop on program analysis for software tools and engineering, pp. 28-34, 2005

[27] Graham S., Kessler P., McKusick M.: gprof: A Call Graph Execution Profiler. In Proceedings of The SIGPLAN '82 Symposium on Compiler Construction (Boston, MA, June 1982), Association for Computing Machinery, pp.120-126

[28] Douglas Gregor, Jaakko Jrvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.

[29] David Abrahams, Carlos Pinto Coelho: Effects of Metaprogramming Style on Compilation Time. 2001 http://users.rcn.com/abrahams/instantiation_speed/

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

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