Automatic Template Instantiation In DIGITAL C++

Avrum E. Itzkowitz
Lois D. Foltan

Automatic template instantiation in DIGITAL C++ version 6.0 employs a compile-time scheme that generates instantiation object files into a repository. This paper provides an overview of the C++ template facility and the template instantiation process, including manual and automatic instantiation techniques. It reviews the features of template instantiation in DIGITAL C++ and focuses on the development and implementation of automatic template instantiation in DIGITAL C++ version 6.0.

The template facility within the C++ language allows the user to provide a template for a class or function and then apply specific arguments to the template to specify a type or function. The process of applying arguments to a template, referred to as template instantiation, causes specific code to be generated to implement the functions and static data members of the instantiated template as needed by the program. Automatic template instantiation relieves the user of determining which template entities need to be instantiated and where they should be instantiated.

In this paper, we review the C++ template facility and describe approaches to implementing automatic template instantiation. We follow that with a discussion of the facilities, rationale, and experience of the DIGITAL C++ automatic template instantiation support. We then describe the design of the DIGITAL C++ version 6.0 automatic template instantiation facility and indicate areas to be explored for further improvement.

C++ Template Facility

The C++ language provides a template facility that allows the user to create a family of classes or functions that are parameterized by type.[1,2] For example, a user may provide a Stack template, which defines a stack class for its argument type. Consider the following template declaration:

```
template <class T> class Stack {
   T *top_of_stack;
public:
   void push( T arg );
   void pop( T& arg );
};
```

The act of applying the arguments to the template is referred to as template instantiation. An instantiation of a template creates a new type or function that is defined for the specified types. Stack<int> creates a class that provides a stack of the type int. Stack<user_class> creates a class that provides a stack of user_class. The types int and user_class are the arguments for the template Stack.

In general, a template needs to be instantiated when it is referenced. When a class template is instantiated, only those member functions and static data members that are referenced are also instantiated. In the Stack example, the member function Push of the class Stack<int> needs to be instantiated only if it is used. Template functions and static data members have global scope; therefore, only one instantiation of each should be in a user's application. Since source files are compiled separately and combined later at link time to produce an executable, the compiler alone is not able to ensure that one and only one instance of a specific template is efficiently generated for any given executable. That is, the compiler by itself is not able to know whether the function or variable definition for a specific template is satisfied by code generated in another object module.

The C++ Standard provides facilities for the user to specify where a template entity should be instantiated.[1] When the user explicitly specifies template instantiation, the user then becomes responsible for ensuring that there is only one instantiation of the template function or static data member per application. This responsibility can necessitate a considerable amount of work. However, the compiler and linker working together can provide effective template instantiation without specific user direction.

In the following section, we present the various approaches that can be used for template instantiation.

Template Instantiation Techniques

Template instantiation techniques can be broadly categorized as either manual or automatic. With manual instantiation, the compilation system responds to user directives to instantiate template entities. These directives can be in the source program, or they may be command-line options. With automatic instantiation, the compilation system, including the linker, decides which instantiations are required and attempts to provide them for the user's application.

Manual Instantiation

Manual template instantiation is the act of manually specifying that a template should be instantiated in the file that is being compiled. This instantiation is given global external linkage, so that references to the instantiation that are made in other files resolve to this template instantiation. Manual template instantiation includes explicit instantiation requests and pragmas as well as command-line options.

Explicit Instantiation Requests and Pragmas The compilation system instantiates those template entities that the user specifies for instantiation. The specification can be made using the C++ explicit template instantiation syntax or may be made using implementation-defined directives or pragmas. Since instantiations are given global external linkage, the user must ensure that the specified template instantiations appear only once throughout all the modules that compose the program. When only this mode of instantiation is used, the user also must ensure that all required template instantiations are specified to avoid unresolved symbols at link time.

Command-line Instantiation Command-line options can be used to specify template instantiation. They are similar in operation to the explicit instantiation requests, except they indicate groups of templates that should be instantiated, rather than naming specific templates to be instantiated. The command-line options include

- Instantiate All Templates. A command-line option can direct the compiler to instantiate all template entities whose definitions are known during compilation and whose argument types are specified. This has the advantage of specifying many template instantiations at once. The user must still ensure that no template instantiation happens more than once in the program and that all required instantiations are satisfied. Due to these requirements, the user cannot usually specify this option on more than one source-file compilation in the program. This option can also cause the instantiation of templates that are not used by the program.
- Instantiate Used Templates. A command-line option can be used to direct the compiler to instantiate only those template entities that are used by the source code and whose definitions are known at compilation. As in the previous technique, the user must ensure that no template instantiation happens more than once in the program and that all required instantiations are satisfied. Due to these requirements, the user cannot usually specify this option on more than one source-file compilation in the program.
- Instantiate Used Templates Locally. This command-line option works like the instantiate used templates option, except that it defines each template instantiation locally in the current compilation. This option has the advantage of providing complete template instantiation coverage for the program, as long as the definitions of the used templates are available in each module. Since all template instantiations are given local scope, there is no potential problem with multiply defined instantiations when the program is linked. The major problem with this technique is that the user's application can be unnecessarily large, since the same template instantiations could appear within multiple object files used to link the application. This technique will fail if the instantiations must have global scope such as a class's static data members.

Figure 1 shows an example of a template function, template_func, that contains a locally defined static variable. As shown in the figure, the object files of both A and B contain local copies of template_func instantiated with int. Each instance of template_func<int> defines its own version of static variable *x*. In this case, directing the compiler to instantiate used templates locally yields a different result than instantiating all or used templates globally.

If we give the static data members global scope and ensure that they are properly defined and initialized by executable code rather than by static initialization, we can solve the static data members problem. The application, however, remains unnecessarily large, because multiple copies of the instantiated templates can be present in the executable.

Automatic Instantiation

Automatic template instantiation relieves the user of the burden of determining which templates must be instantiated and where in the application those instantiations should take place. Automatic template instantiation can be divided into two categories: compile-time instantiation, whereby the decision about what should be instantiated is made at compile time, and link-time instantiation, whereby decisions about template instantiation are made when the user's application is linked. In both cases, specific link-time support is needed to select the required instantiations for the executable.

Compile-time Instantiation Two major techniques can be used to perform automatic template instantiation at compile time. The choice between the two depends upon the facilities available in the linker. Microsoft Visual C++ instantiates templates at compile time using a strategy similar to the instantiate used templates command-line option described previously.[3] Each instantiation is placed in the communal data section (COMDAT) of the current compilation's object file. Each object file contains a copy of every template instantiation needed by that compilation unit. COMDATs are sections that have an attribute that tells the linker to accept, without issuing a warning, multiple definitions of a symbol defined in the section.[4] If more than one object file defines that symbol, only the section from one object file is linked into the image and the rest are discarded, along with all symbols in the symbol table defined in the discarded section contribution. At link time, the linker resolves an instantiation reference by choosing one of the instantiations defined in an individual object file's COMDAT. The resulting user's application executable has a single copy of each requested instantiation.

When such linker support is not available, another mechanism must be used to control compile-time instantiation. One such approach is to use a repository to contain the generated instantiations. The compiler creates the instantiations in the repository instead of the current compilation's object file. At link time, the linker includes any requested instantiations from the repository. As a performance improvement, the compiler can also decide whether an instantiation needs to be generated from the state of the repository. If the requested instantiation is in the repository and can be determined to be up to date, the compiler does not need to regenerate the instantiation.

Link-time Instantiation The decision to instantiate can be left until link time. The linker can find the instantiations that are needed and direct the compiler to generate those instantiations. McCluskey describes one link-time instantiation scheme.[5,6] The compiler logs every class, union, struct, or enum in a name-mapping file in a repository. Every declared template is also logged in the name-

```
//template.hxx
#include <iostream.h>
template <class T> void template_func (T p)
{
   static T x = 0;
   cout << x + p;
   x++;
}

//A.cxx                        //B.cxx
#include "template.hxx"        #include "template.hxx"
extern void b_func();          void b_func(void)
int main()                     {
{                                 //...
   template_func(10);             template_func(20);
   b_func();                      //...
   return 0;                   }
}
```

Figure 1
Template Function Containing a Locally Defined Static Variable

mapping file. At link time, a prelinker determines which template instantiations are required. The prelinker builds temporary instantiation source files in the repository to satisfy the referenced instantiations, compiles them, and adds the resulting object files to the linker input. Consider the example in Figure 2.

During the compilation of main.cxx, a name-mapping file is built in the repository and the location of the user-defined class C and the function template, perform_some_function, are recorded. From the information stored in the name-mapping file, an instantiation source file is then created in the repository. Figure 3 shows the contents of the instantiation source file created to satisfy perform_some_function<C>.

The prelinker then compiles the instantiation source file by invoking the compiler in a special directed mode, which directs the compiler to generate code only for specific template instantiations that are listed on the command line. The compiler then generates the definition of perform_some_function<C> in the resulting object file. The resulting object now satisfies the instantiation request and is included as part of the application's final link. To build the instantiation source files easily, the implementation of this scheme generally requires that template declarations, template definitions, and any argument types used to instantiate a class or function template must appear in separate, related header files.

The Edison Design Group has developed another approach to link-time instantiation.[7] In this approach, the compiler records where template instantiations are used and where they can be instantiated. At link time, a prelinker assigns template instantiations by recording the assignments in a specially generated file that corresponds to the particular source file that can successfully instantiate the user's request. Compiling and prelinking the program used in Figure 2 generates an instantiation assignment file for main.cxx. This file contains information concerning the command-line options specified, the user's current working directory, and a list of instantiations that should be instantiated. Main.cxx now owns the responsibility of instantiating perform_some_function<C>. The prelinker recompiles the source files, such as main.cxx, that have changes in their template instantiation assignments. The process is repeated until there are no changes made to the instantiation assignments. Then the final link can be completed.

This approach has the advantage of requiring no special file structure to support automatic template instantiation. It is generally faster and simpler than McCluskey's approach, because fewer files are compiled in the generation of the needed instantiations and the instantiations are generated in the context of the user's source code. In addition, the assignment of instantiations to source files can be preserved between recompilations of the source code, so that unless the structure of the application changes, the needed instantiations will be available without additional recompilation.

```
/* perform_some_function(C&) */
#include "template.hxx"
#include "template.cxx"
#include "C_class.h"
```

Figure 3
Example of an Instantiation Source File

```
//C_class.hxx
class C {
public:
   //...
        };

//template.hxx
template <class T> void perform_some_function(T &param);

//template.cxx
template <class T> void perform_some_function(T &param) { }

//main.cxx
#include "C_class.hxx"
#include "template.hxx"

int main()
{
    C c;
    perform_some_function(c);
    return 0;
}
```

Figure 2
Example of a Link-time Instantiation Scheme (McCluskey)

Comparison of Manual and Automatic Instantiation Techniques

The manual instantiation techniques require planning on the part of the user to ensure that needed instantiations are present, that no extraneous instantiations are generated, and that each needed instantiation appears exactly once within the application. With manual instantiation, the user has the advantage of gaining explicit control over all template instantiations. Although the strategy of instantiating used templates locally requires less planning, it does so at the cost of object file size and the restricted use of templates when static data members are present or when static data is defined locally within a function template instantiation.

Automatic template instantiation provides template instantiation with no explicit action on the part of the user. Compile-time instantiation requires either specific linker support to select a single template instantiation from potentially many candidates, or support by the compiler to generate instantiations in separate object files while compiling the user's source code. Relying on linker support allows the compiler to efficiently generate instantiations at the cost of larger object files; however, the user loses control over which instantiation is used in the executable file. Although the use of separate instantiation object files usually takes more time at compilation than the linker-support method, it results in more compact object files and can provide the user with more control over which instantiation is used in the executable file.

Link-time instantiation provides template instantiation that is tailored to the needs of the executable file. The primary cost is link-time performance, since generation of instantiations occurs at link time. Another disadvantage of link-time instantiation can be observed when building object-code libraries. Either the library must contain all the instantiations that it requires, or the user who wants to link with the library must have access to all the machinery to create instantiations. Creating a library's instantiations involves extra steps during library construction. All the object files to be included in the library must be prelinked, so that the needed instantiations are generated. If instantiations are included in the individual object files in the library, as in the Edison Design Group approach, unintended modules may be linked from the library to provide the needed instantiations. Consider the following scenario, in which object files A and B are included in the library. Both files require the instantiation of perform_some_function<int>. When these files are prelinked, the instantiation of perform_some_function<int> is assigned to one of the files, say A. If an application that is being linked against the library requires that the object file B be linked into the executable, then the object file A is also linked. Here the instantiation needed by B was instantiated in A even though the executable never referenced anything explicitly defined in file A. This can yield an unnecessarily large executable.

In the next section, we review the template instantiation support in earlier versions of DIGITAL C++ and then discuss the rationale and design of the automatic template instantiation facility in version 6.0 of DIGITAL C++.

DIGITAL C++ Template Instantiation Experience

As the use of C++ templates has grown, DIGITAL C++ has been enhanced to support the need for improved instantiation techniques. The initial release of DIGITAL C++ occurred before the C++ standardization process had matured, so that the language supported was based on *The Annotated C++ Reference Manual,* referred to as the ARM.[8] The ARM defined template functionality, but it did not provide guidance for either manual or automatic template instantiation. Thus it was necessary to provide a DIGITAL C++-specific mechanism for template instantiation.

DIGITAL C++ Manual Template Instantiation

The #pragma define_template directive and the instantiate all command-line option, -define_templates, have been supported since the initial release of DIGITAL C++.

In Figure 4, the define_template pragma directs the compiler to instantiate class template, C, with type int. When the compiler detects the use of the pragma, it creates an internal C<int> type node and traverses the list of static data members and member functions defined within the class. If the definitions of these members are present at the point the pragma is specified, the compiler materializes each with type int.

As the C++ language developed and template usage increased, users found manual template instantiation to be very labor intensive and requested an automated method.

DIGITAL C++ Version 5.3 Automatic Template Instantiation

Automatic template instantiation capability became a serious issue during the planning stages of DIGITAL C++ version 5.3. The use of templates was increasing rapidly, and many new third-party libraries, such as Rogue Wave Software's Tools.h++, contained a significant use of templates. Due to this growing need, the requirements were straightforward. The support had to be easy to use, have a short design phase, be quickly implementable on both the DIGITAL UNIX and the OpenVMS platforms, and provide reasonable performance. Because McCluskey's approach had been used in several implementations, it presented itself as our best option.

```
template <class T> class C {
public:
   void mem_func1(T p);
   void mem_func2(T p);
   ...
};

template <class T> void C<T>::mem_func1(T p) { //...}
template <class T> void C<T>::mem_func2(T p) { //...}

#pragma define_template C<int>
```

Figure 4
The define_template Pragma

DIGITAL made two major changes to McCluskey's approach to take advantage of the DIGITAL C++ compiler design. First, we allowed instantiation source files to be created at compile time instead of link time. This eliminated the need for McCluskey's name-mapping file and simplified the prelinking process considerably. Since the needed source files existed in the repository, there was no need to deconstruct the required template instantiations to determine their arguments and types.

The second change addressed the transitive closure problem. Figure 5 shows an example of the class template Buffer being instantiated with the user-defined type C. After compilation of app.cxx with the McCluskey approach, the name-mapping file contained definition locations of class B and class C. However, it did not contain any indication that class C had a data member that relied on the definition of class B. From the information in the name-mapping file, the prelinker then created an instantiation source file that included only C_class.hxx, Buffer.hxx, and Buffer.cxx. When this instantiation source file was compiled, an error resulted complaining that B is an undefined type whose size is unknown.

We solved this problem in DIGITAL C++ version 5.3 by including all the top-level header files included by the current compilation unit in any instantiation source files created. This ensured that B_class.hxx would be included in the generated instantiation file.

```
//B_class.hxx
class B { //... };
```

```
//C_class.hxx
class C {
   B data_mem;
public:
   //...
};
```

```
//Buffer.hxx
template <class T> class Buffer {

   T *buffer;
   int num_of_items;
public:
   void add_item(T *);
   //...
};
```

```
//Buffer.cxx
template <class T>
         void Buffer<T>::add_item(T *p) { }
```

```
//app.cxx
#include "B_class.hxx"
#include "C_class.hxx"
#include "Buffer.hxx"

void f(void)
{
   C c;
   Buffer<C> c_buffer;
   c_buffer.add_item(&c);
}
```

Figure 5
Instantiation of the Class Template Buffer

Despite the fact that this type of automatic link-time instantiation scheme was being widely used in the industry, the results of using a modified McCluskey approach were mixed. Stroustrup has described the general problems with McCluskey's approach.[9] We found that our implementation suffered particularly from poor link-time performance and so did not satisfy our users' needs.

DIGITAL C++ Version 6.0 Automatic Template Instantiation

DIGITAL C++ version 6.0 is a complete reimplementation of DIGITAL C++, with emphasis on ANSI C++ conformance. It is implemented using a completely new code base, which includes the industry-standard C++ front end from the Edison Design Group and a standard class library from Rogue Wave.

From our experience with template instantiation in DIGITAL C++ versions 5.3 through 5.6, we concluded that the most important issue that should be addressed in the design and implementation of the automatic template instantiation facility was the compile- and link-time performance. The primary goal was to have the performance of automatic template instantiation substantially exceed the performance of version 5.6. Another important goal was to remove the restriction of template declaration and definition placement in header files. In addition, the automatic template instantiation facility in version 6.0 had to be culturally compatible with the previous implementation. The user had to be able to move sources and objects to different directories, easily build archived and shared libraries, share instantiations between various applications, and have error diagnostics reported at the earliest possible moment in the instantiation process.

Design and Implementation We decided to use a compile-time instantiation model as the basis for our implementation. Since we were using the Edison Design Group's front end, we seriously considered using their link-time model. However, the compile-time model seemed advantageous for several reasons. First, there are significant complications (as described in the section Comparison of Manual and Automatic Instantiation Techniques) when trying to build libraries with a compiler that uses the Edison Design Group link-time model. In addition, the link-time model requires recompilations that limit performance in many typical cases of template use. We recognized that the link-time model could provide better performance in some cases, but these would be in the minority. Finally, the implementation of the link-time model would require substantially more implementation effort on the OpenVMS platform. The version of the Edison Design Group front end being used to build DIGITAL C++ version 6.0 required tools to scan a user's object files for information concerning which modules could instantiate requested templates. Similar functionality would need to be implemented for the OpenVMS platform.

We preserved the concept of the template repository as a directory that contains the individual template instantiation object files. The repository stores one object file for each template function, member function, static data member, and virtual table that is generated by automatic template instantiation. The file name of the instantiation object file is derived from the name of the instantiation's external name. At compile time, the front end generates intermediate code for all templates that are needed in the compilation unit and can be instantiated. A tree walk is performed over the intermediate code to find all entities that are needed by each generated template instantiation. The code generator is called to generate code for the user-specified object file and is then called repeatedly for each template instantiation to generate the instantiation object files in the repository.

The compiler generally considers an instantiation to be needed when it is referenced from a context that is itself needed, such as in a function with global visibility or by the initialization of a variable that is needed. Virtual member functions are needed when a constructor for the class is needed. Thus, all virtual function definitions should be visible in a compilation unit that requires a constructor for the class. Each instantiation that is generated with automatic instantiation is marked as potentially being in its own object file in the repository.

The intermediate representation of each generated instantiation is walked to determine what other entities it references. At this point, the instantiation is a candidate to be generated in its own object file, but it can sometimes be generated as part of the user-specified object file. If the instantiation references an entity that is local to the compilation unit, such as a static function, and that local entity is nonconstant and statically initialized, the instantiation is merged into the user-specified object file rather than generated in its own object file. As an alternative, we could have chosen to change the local entity into a global entity with a unique name and generate the instantiation in its own object file. We chose not to do this in order to make it easier to share a repository between applications. With this alternative, the instantiation in the repository requires the object file containing the local entity's definition, which may be in another application. Note that any application that contains more than one definition of the same instantiation that references a nonconstant local entity is a nonstandard-conforming application. This is a violation of the one definition rule.[10] Consider the following code fragment:

```
static int j;
template <class T> int func (T arg) { return j; }
int var = func( 2.5 );
```

The reference to the static variable *j* in the template function, func, prevents the template from being generated into its own object file in the repository.

When the individual instantiations are walked, we mark each global entity that is defined in the compilation unit so that the definition is replaced by an external reference when the instantiation object file is generated. Consider the following code fragment:

```
void print_count(const char * s, int ivar)
{
   cout<< s <<":" << ivar;
}

template <class T> void func (T arg)
{
   static int count = 0;
   print_count("count", count++);
}
```

The function, print_count, is defined in the source file and generated as a defined function in the user-specified object file. The template function, func, references the function, print_count. When the code for func is generated in its own object file, the reference to print_count must be changed from a reference to a defined function to a reference to an external function.

By default, each needed instantiation is generated by every compilation that requires the instantiation. This is the safe default because it ensures that instantiations in the repository are up to date. However, there will probably be some compilation overhead from regenerating instantiations that may already be up to date. We believed that the overhead of regenerating instantiations would typically be relatively small. For applications with a high overhead of instantiation, such as a large number of source files using the same large number of template instantiations, we provided a compilation option to control the generation of template instantiations to improve compile-time performance.

The generation of instantiation object files only when they are actually required is a difficult problem. Fine-grain dependency information would have to be kept for each instantiation object file. Such dependency information would need to reflect those files that are required to successfully generate the instantiation and record which command-line options the user specified to the compiler. We suspected that the overhead involved with gathering and checking the information might be an appreciable percentage of the time it would take to do the instantiation, and thus it would not give us the performance improvement that we wanted.

Instead, we decided to provide an option that allows the user to decide when instantiations are generated. We refer to this as the template time-stamp option, -ttimestamp. When using the time-stamp option, the compiler looks in the repository for a file named TIMESTAMP. If the file is not found, it is created. The modification time of this file is referred to as the time stamp. When generating an instantiation, the compiler looks in the repository to see if the instantiation object file exists. If it does not exist, it is generated. If the file already exists, its modification time is compared to the time stamp. If the modification time is later than the time stamp, the instantiation is assumed to be up to date and is not regenerated. Otherwise, the instantiation is generated. The user can control the generation of instantiation object files by changing the modification time of the TIMESTAMP file.

The time-stamp option would typically be used in a makefile or a shell script that compiles and builds an entire application. Before invoking make or the shell script, the user would make certain that no TIMESTAMP file resided in the repository. This would ensure that each needed instantiation would be generated exactly once during all the compilations done by the build procedure.

Much of the C++ linker support in version 5.6 was reused with only minor modifications for version 6.0. The compiler is presented with a single repository into which the instantiation object files are written. Multiple repositories can be specified at link time, and each can be searched for instantiations that are needed by the executable file. The linker is used in a trial link mode to generate a list of all the unresolved external references. This list is then used to search the repositories to find the needed instantiation files, and the process is repeated until no more instantiations are needed or can be satisfied from the repository. The link then proceeds as any normal link, adding the list of instantiation object files to the list of object files and libraries as specified by the user.

If a vendor is creating a library rather than an executable file, the instantiations needed by the modules in the library can be provided in either of two ways: (1) The library vendor can put the needed instantiations in the library by adding the files in the repository to the library file. (2) The library vendor can provide the repository with the library and require that library users link with the repository as well. Note that instantiations placed in the library are fixed when the library is created. Since the library is included in the trial link of an application, any instantiation in the library takes precedence over the same named instantiation in a repository.

Results In a number of tests, DIGITAL C++ version 6.0 showed improved performance over version 5.6. We tested a variety of user code samples that use templates to varying degrees and found that build times for version 6.0 decreased substantially compared to the version 5.6 compiler. Examples of two typical C++ applications used in our tests are the publicly available EON ray-tracing benchmark and a subset of tests from our Standard Template Library (STL) test suite. For

the EON benchmark, the build time for version 6.0 was reduced to 28 percent of the build time for version 5.6. For the STL tests, the build time for version 6.0 was reduced to 19 percent of the build time for version 5.6. The number of files in the repository also decreased significantly because version 6.0 generates only instantiation object files instead of the instantiation source, command, dependency, and object files of version 5.6. For EON, the version 6.0 repository contained 88 files compared to 260 files in version 5.6.

Using the time-stamp option, build time for the EON benchmark was reduced by only 5 percent compared to the default instantiation strategy. The real benefit of the time-stamp option comes with applications that use the same template instantiations in many compilation units. For example, in one user's test case, build times dropped from roughly 18 hours with the default instantiation to 3 hours when using the time-stamp option.

In the next section, we conclude our paper with a discussion of further work that can improve the performance and usability of automatic template instantiation.

Future Research

We continue to investigate approaches and techniques to improve the usability and performance of the automatic template instantiation facility. Optimal usability and performance would seem to require a development environment completely integrated for C++. This environment would keep track of all entity definitions and usage and would be able to limit all instantiation generation to the minimum needed. This approach would require a great deal of development work and might be difficult to integrate with existing customer development methodologies. Therefore, we focus on more modest techniques that approximate the optimal case.

We are exploring ways to improve both performance and usability in the management of dependency information. We continue to look at approaches for using dependencies that can be reliable, automatic, and fast. We also continue to investigate ways to gather and check fine-grained dependency information for the instantiation object files, though performance is a concern. One approximation to the fine-grain dependency information that we are investigating is a larger grain dependency scheme. This technique creates a time stamp from the latest creation time of any source file included during compilation of a given module. Any instantiation object file in the repository whose modification time is later than this time stamp would not be regenerated. This approach is more automatic and can potentially yield better performance than our current time-stamp option, but it would not be sensitive to changes on the command line or changes to the structure of the files used to generate the instantiation. For example, if the user specified an include directory of old_include on the initial compilation and later specified an include directory of new_include, this approach would not recognize that different files were being included.

Another approach to improving application build performance is to support a build facility that can make use of template information in determining dependency. Currently, each user-specified object file is dependent on all the included files necessary to create instantiation object files for template requests. When a change is made to a template definition, all the sources that reference the template need to be recompiled. A build facility designed to be sensitive to template instantiation could detect that a change in the template definition was limited to the instantiation object file. It could then instruct the compiler to suppress the regeneration of object files for source files that are only being recompiled due to the change in the template instantiation. Such a facility could also suppress the recompilation of any source file that would only reproduce the changes to instantiations that were already regenerated.

Because we recognize that link-time instantiation can perform better in some cases than the compile-time approach, we are investigating the link-time instantiation model as a user option.

Finally, we continue to look at ways to reduce the cost of generating each instantiation. For example, by default the compiler compresses the generated object files. Although most instantiation object files are small, many of them are potentially generated in a single compilation. As a result, the time to compress all the instantiation object files can be significant. Improvements such as not compressing small object files and/or improving the algorithm of the object file compression implementation itself could yield significant performance improvement. In addition to improvements that would reduce the overhead of generating instantiations, we are also researching ways to reduce the number of instantiation object files. For example, we might combine all the virtual functions of a class into a single instantiation object file in the repository.

Summary

As with most engineering problems, no single approach to the automatic instantiation of templates is optimal for all potential uses of templates. Based on our experience with providing template support in DIGITAL C++, we chose to implement a compile-time automatic template instantiation scheme for version 6.0 that generates instantiation object files into a repository. This choice allows users to better control when template instantia-

tion occurs. In addition, it provides a substantial improvement in performance of template instantiation over version 5.6 and reduces the restrictions on the location of template declarations and definitions. We continue to investigate the template-instantiation implementation to further improve compile- and link-time performance and ease of use.

Acknowledgment

The authors wish to acknowledge Bevin Brett, who contributed substantially to the design and implementation of the needed walk and instantiation object file generation for DIGITAL C++ version 6.0, and Hemant Rotithor, who provided the performance measurements for DIGITAL C++ version 6.0 versus version 5.6. The authors also wish to acknowledge Charlie Mitchell, Coleen Phillimore, Rich Phillips, and Harold Seigel for their contributions to the design and implementation of the DIGITAL C++ automatic template instantiation.

References

1. ISO/IEC Standard 14882, Programming Language C++, 1998.
2. B. Stroustrup, *The C++ Programming Language,* Third Edition (Reading, Mass.: Addison-Wesley, 1997).
3. Microsoft Visual C++ 5.0, On-line Help, "Templates, C++."
4. Microsoft Corporation, "Microsoft Portable Executable and Common Object File Format Specification," Revision 5.0, Section 5.5.6, *Microsoft Developer's Network* (October 1997).
5. G. McCluskey, "An Environment for Template Instantiation," *The C++ Report,* vol. 4, no. 2 (1992).
6. G. McCluskey and R. Murray, "Template Instantiation for C++," *Sigplan Notices,* vol. 27, no. 12 (1992): 47–56.
7. Edison Design Group, "Template Instantiation in the EDG C++ Front End," Note to the ANSI C++ Committee, X3J16/95-0163, WG21/N0763.
8. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual* (Reading, Mass.: Addison-Wesley, 1990).
9. B. Stroustrup, *The Design and Evolution of C++* (Reading, Mass.: Addison-Wesley, 1994): 366.
10. B. Stroustrup, *The C++ Programming Language,* Third Edition (Reading, Mass.: Addison-Wesley, 1997): 203–205.

Biographies

Avrum E. Itzkowitz
Avrum Itzkowitz was a contractor/consultant at DIGITAL from September 1995 through December 1997. During that time, he worked as part of the DIGITAL C++ development team, designing and implementing much of the support for the automatic template instantiation facility in DIGITAL C++ version 6.0. Avrum also designed and implemented template instantiation tests. He is currently a senior software architect engineer at GTE Internetworking. He holds a B.S. (1972) in electrical engineering from Northwestern University and M.S. (1976) and Ph.D. (1979) degrees in computer science from the University of Illinois. Avrum is a member of the ACM, the IEEE-Computer Society, and SIGPLAN.

Lois D. Foltan
Lois Foltan is a principal software engineer at Compaq. Her areas of expertise include support for C++ automatic template instantiation and the DIGITAL C++ object model. She was a member of the DEC C/C++ compiler team for eight years. During that time, she contributed to the first GEM-based DEC C and DEC C++ compilers. Recently, she joined the Digital Java team. Lois received a B.S. in computer science from the University of Vermont in 1988.