Scientific Programming 11 (2003) 321–327
IOS Press

Dynamic compilation of C++ template code

Martin J. Cole and Steven G. Parker
Scientific Computing and Imaging Institute, School of Computing, University of Utah, Salt Lake City, UT 84112
E-mail: mjc@sci.utah.edu, sparker@cs.utah.edu; URL: www.cs.utah.edu/sci

Abstract. Generic programming using the C++ template facility has been a successful method for creating high-performance, yet general algorithms for scientific computing and visualization. However, adding template code tends to require more template code in surrounding structures and algorithms to maintain generality. Compiling all possible expansions of these templates can lead to massive template bloat. Furthermore, compile-time binding of templates requires that all possible permutations be known at compile time, limiting the runtime extensibility of the generic code. We present a method for deferring the compilation of these templates until an exact type is needed. This dynamic compilation mechanism will produce the minimum amount of compiled code needed for a particular application, while maintaining the generality and performance that templates innately provide. Through a small amount of supporting code within each templated class, the proper templated code can be generated at runtime without modifying the compiler. We describe the implementation of this goal within the SCIRun dataflow system. SCIRun is freely available online for research purposes.

1. Problem description

SCIRun[1] is a scientific problem solving environment that allows the interactive construction and steering of large-scale scientific computations [1–3]. A scientific application is constructed by connecting computational elements (modules) to form a program (network). This program may contain several computational elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. Geometric inputs and computational parameters may be changed interactively, and the results of these changes provide immediate feedback to the investigator. SCIRun is designed to facilitate large-scale scientific computation and visualization on a wide range of machines from the desktop to large shared-memory and distributed-memory supercomputers.

At the heart of any general visualization system is the data model. The data model is responsible for representing a wide range of different data representation schemes in a uniform fashion. In the case of SCIRun, the core piece of our data model is the field library [4, 5], where a field is simply a function represented over some portion of 3D space. In most cases, that function is represented by some discrete approximation, such as a tetrahedral grid (i.e. a Finite Element Mesh) or a 3D rectangular grid (i.e. a Finite Difference mesh, or the product of a 3D medical scan such as Computed Tomography or Magnetic Resonance Imaging). Representing each of these fields in the most general form possible would lead to a number of inefficiencies, including a massive data explosion.

Therefore, we turn to C++ for mechanisms of providing access to these different field types in a uniform way. Typical operations include computing the minimum or maximum value in the field, iterating over discrete data points, and interpolating the value at a specified point in space. In C++, we can use inheritance and virtual functions to maintain a uniform interface to these disparate representations.

We compared the runtime performance in a simple yet representative test program. The test times virtual method calls vs. template method calls of an identical function. The results demonstrate the widely-held knowledge that there can be a significant performance penalty to using a virtual interface. On Linux, the virtual method timed at 30.43 seconds, and the template version at 10.1 seconds, on Irix the same test ran at

[1]Pronounced "ski-run." SCIRun derives its name from the Scientific Computing and Imaging (SCI) Institute at the University of Utah.

ISSN 1058-9244/03/$8.00

10.21 seconds, and 2.84 seconds respectively. These results have led us away from a virtual interface in many cases. Instead, we turn to generic programming. Table 1 shows these results in tabular form.

The C++ template facility has been used by numerous researchers to create high-performance, yet general algorithms for scientific computing and visualization [6–8]. Generic programming relies on the compiler to generate specialized instances of particular algorithms that are tailored to the underlying data representation. However, the use of templated code typically leads to propagation of more template code. Compiling all possible expansions of these templates can lead to massive template bloat. Furthermore, compile-time binding of templates requires that all possible permutations be known at compile time, limiting the runtime extensibility of the generic code.

As an example, consider the following realistic example from the SCIRun field library. Consider the set of different field classes: TetVol (Tetrahedral Volume Grid), LatticeVol (3D Rectangular Lattice Grid), ContourField (A set of countour lines), and TriSurf (A 3D triangulated surface). On each of these fields, we can hold several different types of data, such as double, int, char, unsigned char, unsigned short, short, bool, Vector (3 doubles indicating a direction), and Tensor (6 doubles).

For these four field types, and these nine primitive types, the compiler would be required to generate a total of 36 different field combinations. Now consider these 36 types that are used with a computational or visualization algorithm that is parameterized on one field type. The compiler would also generate 36 versions of this algorithm. However, if the algorithm required two fields, and was therefore parameterized on two different field types, the compiler would be required to generate $36^2 = 1296$ different versions of that algorithm. For an algorithm with three different field types, $36^3 = 46656$ fully instantiated classes would be generated. These numbers grow as more field types, data types, and algorithms are supported.

Our compilers did indeed have problems compiling a fully instantiated version of our code. The compiler itself ran out of 32 bit address space during a global optimization pass. At this point, the template bloat moved from an annoyance to a critical bug.

Since SCIRun is an interactive system, any of these combinations could be used at any time. However, a typical user will use only a handful of different field types while using SCIRun. SCIRun is also extensible at run-time through the dynamic loading of new modules. In particular, new field types can be created by loaded modules, and these fields can be sent to other, pre-compiled modules. With a pure template-based approach, modules that were compiled without support for the new field would not be able to operate.

A different design of the field classes could easily solve this problem, but would have other weaknesses. If we used virtual functions instead of generic programming to access the different types of fields, the system would not suffer from the combinatoric explosion of templated types. However, this design would suffer a different drawback, namely performance. Virtual function calls are costly, and therefore prohibit fine-grained access to data elements. Furthermore, virtual functions thwart many of the optimizations performed by compilers, leading to substantially reduced performance over the template-based approach. As SCIRun is designed for computation and visualization of large-scale scientific datasets, we have found the virtual function solution to be unacceptable in many design situations.

2. Proposed solution

Through the use of C++ templates, the compiler creates multiple versions of the code specific to particular data structures, primitive types, and algorithms. Each module that needs to work on one of the above mentioned classes, implements an algorithm templated on the exact field type. It is this algorithm that gets compiled when it is needed. For the purposes of illustration, consider templates of this form:

```
template<class Field1, class
Field2> class Algorithm;
```

The system operates in the following simple steps:

1. Use C++ RTTI and additional run-time information to determine which field classes are in use. The calling module also specifies which algorithm is to be applied to these fields.
2. Generate a small amount of C++ code to instantiate the correct algorithm with the discovered field types.
3. Fork a process to compile the C++ code into a shared library.
4. Dynamically link this shared library into the running process, and locate the function that will create the instantiated object.
5. Call this function to create an instance of the specialized algorithm.

Table 1
Performance comparison for typical visualization queries using virtual functions and templates

Machine	Compiler	Processor	Virtual function time	Template time
SGI Origin 2000	MIPSPro 7.3.1.2	250 Mhz R10000	10.21 s	2.84 s
Linux PC Pentium III	GNU g++ 3.0	750 Mhz	30.43 s	10.1 s

6. Make a single virtual function call to the algorithm, passing Field1 and Field2 and a generic base class.
7. Since the algorithm knows the concrete type of Field1 and Field2, it uses dynamic_cast to get a pointer to the specific type.
8. Finally, the algorithm performs its operation on the data.

To accomplish this, the algorithm, and the templated classes need to provide some information to the DynamicLoader so that it can create the C++ file that needs to be compiled. Below we explain the mechanisms that are necessary in the code to support these operations.

3. Related work

Kennedy and Syme [9] describe their implementation of generics in the .NET Common Language Runtime. Their work provides a similar solution to the problem of bloat. They use JIT compilation to produce the object at run time, an option enabled by control over the virtual machine. This control enables a faster compile time, as well as the fact that the mechanism is hidden from the user. Essentially we have implemented a crude JIT compilation mechanism for C++. Our compilation/link/load takes longer, but since future runs need not compile and link, the cost is amortized over multiple executions of the SCIRun environment.

POOMA [10] is a high-performance C++ toolkit for parallel scientific computation that depends heavily on C++ templates for achieving high performance code. However, with POOMA, all required templates are instantiated at compile/link time instead of dynamically. Since POOMA is not an interactive system, it does not suffer from some of the same problems as SCIRun; the compiler only generates template instantiations that are required by the scientific program instead of every possible combination. Nevertheless, many POOMA compiles can take considerable time, and some of the template instantiations may never get executed. POOMA does provide constructs beyond what are required for SCIRun, including semi-automatic data parallelism for array expressions and other features. It is possible that our mechanisms could be combined with the expression template engine (PETE) from POOMA in order to provide dynamic compilation of complex scientific simulations.

Veldhuizen [11] describes five different models for compiling C++ template code. Four of these models allow for template instantiation at run-time. However, these mechanisms require compiler and language runtime support not yet found in commercial compilers. In constrast, our system uses commericial and free compilers without modification.

4. Implementation

Through a small amount of supporting code within each templated class, the proper templated code can be generated at runtime. The system generates a small amount of C++ code that includes:

- All C++ header files required to compile the algorithm.
- Any required "using namespace" statements.
- A creation function with "C" linkage that returns an instance of the desired algorithm.

An example of such code is shown in Fig. 1.

4.1. Algorithm structure

A templated algorithm inherits from an algorithm base class. This class defines the interface that the algorithm should have. Each templated algorithm provides the underlying implementation for the pure virtual functions from the base class. The interface has no restrictions, save that it be virtual. All access to the interface happens at the algorithm base class level. Typically the interface is a single pure virtual method with arguments that satisfy the passing of data from the calling module. This allows the entire algorithm to be executed with a single virtual method call. All such algorithm base classes inherit from a common base class that the DynamicLoader maps to the string representation of the exact type for an algorithm.

An example of the algorithm structure is shown in Fig. 2.

```
// This is an automatically generated file, do not edit!
#include "../src/Core/Datatypes/TetVol.h"
#include "../src/Core/Algorithms/Visualization/RenderField.h"
using namespace SCIRun;

extern "C" {
RenderFieldBase* maker() {
  return scinew RenderField<TetVol<double> >;
}
}
```

Fig. 1. An example of the small automatically generated C++ code to instantiate the proper templated class. This will generate the RenderField algorithm, with the field of type TetVol<double>.

```
class TransformScalarDataAlgo : public DynamicAlgoBase
{
public:
  virtual FieldHandle execute(FieldHandle src, Function *f) = 0;

  //! support the dynamically compiled algorithm concept
  static CompileInfo *get_compile_info(const TypeDescription *fsrc,
       const TypeDescription *lsrc);
};


template <class FIELD, class LOC>
class TransformScalarDataAlgoT : public TransformScalarDataAlgo
{
public:
  //! virtual interface.
  virtual FieldHandle execute(FieldHandle src, Function *f);
};


template <class FIELD, class LOC>
FieldHandle
TransformScalarDataAlgoT<FIELD, LOC>::execute(FieldHandle field_h,
     Function *f)
{
  FIELD *ifield = dynamic_cast<FIELD *>(field_h.get_rep());
  FIELD *ofield = ifield->clone();
  typename FIELD::mesh_handle_type mesh = ifield->get_typed_mesh();

  typename LOC::iterator itr, eitr;
  mesh->begin(itr);
  mesh->end(eitr);
  while (itr != eitr)
  {
    typename FIELD::value_type val;
    ifield->value(val, *itr);
    double tmp = (double)val;
    val = (typename FIELD::value_type)(f->eval(&tmp));
    ofield->set_value(val, *itr);
    ++itr;
  }

  return ofield;
}
```

Fig. 2. An example of an algorithm that applies a function to all the scalar data within a field.

4.2. The TypeDescription object

Each object that supports dynamic compilation, must provide a TypeDescription object. This object holds strings that describe its type, the namespace that it belongs to, and the path to the .h file that declares it. The latter is frequently provided by simply returning the value of the standard __FILE__ preprocessor macro.

Most of this internal type information is not available through the standard C++ RTTI facility, so TypeDescription provides that augmented internal information. This object can also recursively contain the TypeDescriptions for sub types. For example a foo<bar, foobar<int> > has a TypeDescription that has a both bar, and foobar TypeDescriptions. The foobar TypeDescription, has the int TypeDescription. A recursive traversal of this object allows us to output a string that matches the exact type for the object.

4.3. The CompileInfo object

Instantiated algorithms have the form Alg<T>, where Alg is the generic algorithm and T a type parameter. A module that wants to create such an algorithm can not have an instance of the algorithm until after dynamic compilation. For this reason, the CompileInfo object is needed. It provides information similar to the TypeDescription, but without the mapping to an underlying object. This object is also the structure that ultimately holds the strings that get written to the .cc file in preparation for compilation. The CompileInfo gets filled with its information when it is passed along to each TypeDescription object that makes up the data type, as well as to the algorithm. The completed CompileInfo object is passed to the DynamicLoader when the calling module requests an instance of the specialized algorithm.

4.4. The DynamicLoader object

The DynamicLoader is the interface for a module to get a handle for the algorithm it needs. Its interface is simple: A module builds up a CompileInfo for the module, and asks the DynamicLoader for a handle to algorithm object. The DynamicLoader then looks up the algorithm in an internal cache. If it does not exist, it uses the CompileInfo to write a small C++ file to disk in a predefined directory. This directory has a makefile that knows how to build a shared library from that C++ file. The DynamicLoader then forks a shell and builds the desired library. Once the shared library is compiled, it is loaded and stored in the internal cache. Each dynamically compiled library has a uniformly named creation function, maker(), which returns a pointer to the algorithm base class. This function pointer is stored in the hash table, and called each time an algorithm is requested by a module, giving each module a separate instance of the algorithm, including unique state for each algorithm instance. Since SCIRun is a multi-threaded program, the DynamicLoader has synchronization code designed such that threads block waiting for a unique type, but it can compile an unlimited number of distinct algorithms concurrently.

4.5. Calling module

The calling module knows of the DynamicLoader, and has a concrete Field instance for which an algorithm template must be instantiated. The algorithm base type is known, as it is integral to the module's function. The exact algorithm will be templated on the exact Field type. This is only known to the module through strings, not types. The module fetches the CompileInfo from the algorithm base class, by feeding it the input Field's TypeDescription object, then asks the DynamicLoader for an algorithm that matches the CompileInfo. No instance of the exact types are instantiated until runtime when they are asked for by the module.

An example of how a module gets and calls an algorithm is shown in Fig. 3.

4.6. Performance

SCIRun is currently supported on Irix and Linux. The runtime compilation and linking depends of course on the code size of the algorithm, but it is typically on the order of seconds. For a user who is not modifying the code that the algorithm depends upon, this is a one time operation. The library remains on disk, so that upon the next run the library can simply be reloaded after a makefile-based dependency check, skipping the compile step.

For a commonly used library in the SCIRun system, the initial compile and link step takes about 7 seconds on Linux, and about 40 seconds on Irix. It should be noted that the longer Irix compilation and linking often produces better optimized code.

Since the compilation only happens once, the system rapidly amortizes the cost of the compilation from the increased performance during execution of the algorithm. Furthermore, the system facilitates more rapid development cycles, as the typical developer does not need to wait for the compiler to instantiate a multitude of template classes at link time.

```
const TypeDescription *ftd = ifieldhandle->get_type_description();
const TypeDescription *ltd = ifieldhandle->data_at_type_description();
CompileInfo *ci = TransformScalarDataAlgo::get_compile_info(ftd, ltd);
DynamicAlgoHandle algo_handle;
if (! DynamicLoader::scirun_loader().get(*ci, algo_handle))
{
  error("Could not compile algorithm.");
  return;
}
TransformScalarDataAlgo *algo =
  dynamic_cast<TransformScalarDataAlgo *>(algo_handle.get_rep());
if (algo == 0)
{
  error("Could not get algorithm.");
  return;
}

// Create a FieldHandle from the Field pointer returned from algo.
FieldHandle ofieldhandle(algo->execute(ifieldhandle, function));
```

Fig. 3. An example from a calling module. It builds up a CompileInfo, and asks for an algorithm of the appropriate type from the DynamicLoader. After error checking it calls into the newly loaded algorithm.

4.7. Disadvantages

This system is not built into the language, so it requires source code, and a C++ compiler on the system. There is additional code maintenance required. We require information that the standard C++ RTTI facility lacks. This information (include files, template parameter types, etc) needs to be added to each new field type or algorithm that is added to the system. This is typically only a few lines of simple code, and is not a major burden.

There are other also a few other minor disadvantages with the current implementation. The libraries are all created in a single directory, so users sharing a build must have write permissions in the directory. The runtime compilation step can be time consuming for a large network of modules the first time through. Developers may not see compile errors until runtime, when the actual instantiation of the exact algorithm gets compiled.

5. Future work

The SCIRun dynamic compilation framework has been used for instantiating classes that could have been known at compile time. We could achieve higher performance in some cases by using even more run-time information in the dynamic compilation phase. For example, array dimensions or repeatedly used constant values could be compiled into the template instance to achieve higher performance.

The current system does not provide a mechanism for specifying special libraries that an algorithm or field class may need. As a result, the makefiles link the shared object against several known libraries, many of which may not be needed. This deficiency could be overcome by requiring the developer to specify required libraries in the TypeDescription and CompileInfo objects, or by assuming that the libraries have previously been loaded into the running program

SCIRun operates under a shared-memory parallel environment. In this case, we are only required to synchronize demand-compilation within a single process. Future versions of SCIRun will operate in a distributed-memory parallel environment, which will require that multiple processors synchronize to avoid race conditions when generating code on a single shared filesystem. In this case, the locking mechanism mentioned above will be extended to use filesystem-based locks.

6. Summary

We have provided a mechanism for compiling only the template instantiations that are needed as opposed to compiling all possible combinations of instantiations. This solution minimizes the biggest problem with using template code, namely bloat: compiling all possible combinations of template code, increases total space and compilation time requirements. This reason has been enough to overshadow the benefits that templates provide in generality and execution time. The deferred compilation scheme makes the use of templates practical for an interactive, general purpose system such as

SCIRun. This mechanism also allows SCIRun modules to operate on data types that it knows nothing about at the time the module is compiled.

References

[1] S.G. Parker, D.M. Beazley and C.R. Johnson, Computational steering software systems and strategies, *IEEE Computational Science and Engineering* **4**(4) (1997), 50–59.

[2] S.G. Parker, *The SCIRun Problem Solving Environment and Computational Steering Software System*, PhD thesis, University of Utah, 1999.

[3] S.G. Parker and C.R. Johnson, SCIRun: A scientific programming environment for computational steering, in *Supercomputing '95*, IEEE Press, 1995.

[4] S.G. Parker, D.M. Weinstein and C.R. Johnson, The SCIRun computational steering software system, in: *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen, eds, Birkhauser Press, 1997, pp. 1–44.

[5] P.J. Moran and C. Henze, Large field visualization with demand-driven calculation, in *Visualization '99*, IEEE Press, 1999.

[6] T.L. Veldhuizen and M.E. Jernigan, Will C++ be faster than Fortran? in *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments* (*ISCOPE'97*), Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[7] S. Haney, J. Crotinger, S. Karmesin and S. Smith, Pete: Portable expression template engine, 1998.

[8] E. Johnson and D. Gannon, Programming with the hpc++ parallel standard template library, 1997.

[9] A. Kennedy and D. Syme, Design and implementation of generics for the .NET common language runtime, in *Programming Language Design and Implementation*, ACM Press, 2001, pp. 1–12.

[10] S. Atlas, S. Banerjee, J. Cummings, P. Hinker, M. Srikant, J. Reynders and M. Tholburn, Pooma: A high performance distributed simulation environment for scientic applications, 1995.

[11] T.L. Veldhuizen, Five compilation models for C++ templates, in *First Workshop on C++ Template Programming,* Erfurt, Germany, 10 October 2000.

Martin Cole

Martin Cole is the Software Manager for the BioPSE development effort. BioPSE is a software tool built within the SCIRun Software System, for the purpose of bioelectric field modeling, simulation, and visualization. He received his B.S. in Computer Science in 1994, and went on to work for Parametric Technology Corporation, specifically on the 3DPaint and CDRS Software systems.

Steven Parker

Steven Parker is a Research Assistant Professor in Scientific Computing and Imaging (SCI) Institute in the School of Computing at the University of Utah. His research focuses on problem solving environments, which tie together scientific computing, scientific visualization, and computer graphics. He is the principal architect of the SCIRun Software System, which formed the core of his Ph.D. dissertation, and is currently the chief architect of Uintah, a software system designed to simulate accidental fires and explosions using thousands of processors. He was a recipient of the Computational Science Graduate Fellowship from the Department of Energy. He received a B.S. in Electrical Engineering from the University of Oklahoma in 1992, and a Ph.D. from the University Utah in 1999.