FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Fast Expression Templates - Enhancements for High Performance C++ Libraries

Jochen Härdtlein, Alexander Linke and Christoph Pflaum

Lehrstuhlbericht 06 - 7

Fast Expression Templates - Enhancements for High Performance C++ Libraries

Jochen Härdtlein, Alexander Linke and Christoph Pflaum

Abstract

Expression templates (ET) can significantly reduce the implementation effort of mathematical software. For some compilers and expressions, however, it can be observed that classical ET implementations do not deliver the expected performance. This is because aliasing of pointers in combination with the complicated ET constructs becomes much more difficult. Therefore, we introduced the concept of enumerated variables, which are provided with an additional integer template parameter. Based on this new implementation of ET we obtain a C++ code whose performance is very close to the handcrafted C code. These so-called Fast Expression Templates (FET) use the principle of enumerating variables to assist the compiler in optimization.

1 The History of Expression Templates

Veldhuizen and Vandevoorde concurrently recognized the potential of expression templates (ET) in the C++ programming language. In 1995 and 1996 Veldhuizen published his first articles about template meta programming [10] and ET [11]. ET were proposed to overcome performance problems which arise from simple operator overloading in mathematical expressions. They avoid unnecessary temporaries by performing some kind of expression dependent inlining within a single loop. Therefore, less data is pushed through the memory hierarchy during the evaluation. For many systems the efficiency of ET implementations competes with their Fortran counterparts [12], but unfortunately not for all. Performance problems with ET were studied for the first time by Bassetti, Davis, and Quinlan in 1997, see [1] and [2]. The main problem turned out to be that compilers have difficulties to resolve the complicated template constructs in the designated way. We will discuss this issue in more detail below. The second obvious problem is caused by the additional storage requirements of the expression parts in the wrapper classes. These two reasons cause - mainly for small data sets - the performance losses in the usage of ET. In the following we will present the classical implementation of ET, analyze a simple example and compare the result with the corresponding handcrafted C code. After that we will explain two different implementations of the concept of Fast Expression Templates (FET) which is based on the idea of enumerated variables. We will provide several examples to confirm the performance expectations.

2 ET introduced by Veldhuizen

For the implementations presented below we always use the same example, the component-wise multiplication of vectors. The parts for this example are the `class Vector` representing the operating class for the vector data, the `class Times` which provides the multiplication of vectors or expressions, and the overloaded `operator *` that calls the appropriate methods. The interior evaluation calls the function `give(int i)`, that has to be implemented in each operating class and the vector class. This somewhat cryptic notation of the classes and functions serves for the compact design of the code fragments below. The operating principle of ET was published by Veldhuizen in [10]. In the following Listing this idea is applied to our example.

```
template<typename L, typename Op, typename R>
struct X{
  const L& l;   const R& r;
```

```
  X(const L& l_, const R& r_) : l(l_),r(r_){}
  double give(int i) const {return Op::apply(l.give(i), r.give(i));}
};

struct Times{
   static double apply(double a, double b) {return a*b;}
};

class Vector{
  double *data;  const int N_;
 public:
  Vector(int N, double val);
  double give(int i) const {return data[i];}
  template<typename L, typename Op, typename R>
  void operator=(const X<L, Op, R>& x){
    for(int i=0; i < N_; ++i) data[i] = x.give(i); }
};

template<typename L>
X<L,Times, Vector> operator*(const L& l,const Vector &v){
  return X<L,Times, Vector>(l,v);
}
template<typename L, typename R>
X<L,Times, R> operator*(const L& l,const R &r){
  return X<L,Times, R>(l,r);
}
```

The wrapper class `X` connects two expressions `L` and `R` by an operation `Op`. `l` and `r` represent the left and the right child in the template parse tree, respectively. The evaluation is done in the function `give`, that starts the call of the static member function `apply(...)` of the operating class `Op`. The core idea is that the `operator *` does not evaluate the operation but only returns an object that stores the expression information as a template type. The evaluation is only started in the assignment `operator =` of the class `Vector` and thus the expression is, in principle, inlined and computed in one loop. Let us study this in more detail by considering the example `c = a*b*a`. An expansion of the type inference mechanism would look as follows:

```
c = a*b*a;
  = X<X<Vector,Tmes,Vector>,Times,Vector>(X<Vector,Times,Vector>(a,b),a)
  =: x
```

Here, we denote the template tree build by the expression `x`. Now we present the call of the assignment `operator =` of the vector `c` for this expression. This leads directly to the loop that evaluates the expression for each index `i` and invokes `x.give(i)`:

```
c.operator= <X<X<Vector,Times,Vector>,Times,Vector> > >(x){
              for(int i = 0; i < N_; ++i) data[i] = x.give(i); }
```

Next, let us concentrate on the call `x.give(i)` for a fixed index `i`. An expansion of this shows the call of `Times::apply(...)` of the operating class and `give(i)` of the corresponding data pointers.

```
c.data[i] = x.give(i);
 = Times::apply(x.l.give(i),x.r.give(i));
 = Times::apply(Times::apply(x.l.l.give(i),x.l.r.give(i)),x.r.give(i));
 = Times::apply(Times::apply(x.l.l.data[i],x.l.r.data[i]),x.r.data[i]);
 = x.l.l.data[i] * x.l.r.data[i] * x.r.data[i];
```

Using the names of the pointers they denote in fact we obtain a code that resembles to a native implementation in C:

```
for(int i=0; i < N; ++i)  c.data[i]=a.data[i]*b.data[i]*a.data[i];
```

This is the principle how ET should work. But as we will subsequently demonstrate compilers have problems with this ET approach.

3 Performance Problems

This classical implementation of ET does not perform for every expression in the manner explained above. These problems with ET were studied for the first time by Bassetti, Davis, and Quinlan in 1997, see [1] and [2]. For the sake of simplicity, let us consider the computation from above, `c=a*b*a`. On an Intel Pentium 4 machine, for instance, this compiled with the g++-3.3.3 compiler achieves a performance inferior to handcrafted C code. Bassetti, Quinlan, and Davis figured out that every leaf in the expression parse tree has its own different data pointer, even the same vectors containing the same data. The compiler does not cope with `x.l.l.data` and `x.r.data` representing the same pointer `a.data`. In principle this is a problem of aliasing. As a consequence, for each reference to an entry of the array of vector a the data is loaded, at the second time starting a load instruction, while it is already in cache. This obviously causes performance lacks and prevents arithmetical optimizations. Since the repeated use of the same vectors in one expression is rather unusual, this problem only appears for some applications. On platforms that work with vectorization aliasing is even more important. Concurrently computations of array entries need very accurate knowledge about pointer dependencies. The performance results reached by ET code affirm this problem. This is one important reason why C++ codes are rarely used on such platforms.

For small vector lengths another problem appears. In each instance of the wrapper class X the left and the right children have to be stored. This should be needless since the template argument contains all information for the computation. This extra storage affects the performance of ET for small vector lengths.

4 Alternative Implementation of ET

Another policy of implementing ET is presented in Listing below.

```
template<class A> class X{
 const A& cast() const {return static_cast<const A&>(*this);}
};

template <class L class R>
struct Times : public X<Times<L,R> > {
  const L& l; const R& r;
  Times(const L& l_, const R& r_) : l(l_), r(r_){}
  double give(int i) const {return l.give(i)*r.give(i);}
};

class Vector : public X<Vector> { ...
  template<class A>
  void operator=(const X<A>& a){
   for(int i = 0; i < N_ ; ++i)
     data[i] = a.cast().give(i);}
};
template <class L, class R>
inline Times<L,R> operator*(const X<L>& l, const X<R>& r){
  return Times<L,R>(l.cast(),r.cast());
}
```

The wrapper class `X` only contains a method for a static cast to its template class type which is always the child of the wrapper class in the parse tree. `X` therefore serves as an interface for the occurring operating classes which inherit from it. The wrapper class `X` only encloses the whole expression and does not appear inside the parse tree anymore. This enclosing `X` is removed by a cast while combining two expressions in the overloaded operator functions similar to the Barton-Nackman-Trick, see [13]. The rest of this implementation works analogously to the presentation above. This implementation provides no improvement in terms of performance but we prefer it because the wrapper class does no longer appear inside the expression tree.

5 Enumerated Variables

As a basis for the FET implementations below an approach for distinguishing the types of variables is necessary. The information the compiler can get from the template parameter of the expression is the combination of vectors but not which vectors occur. In some simplified view the compiler only gets the following type information of the example `c = a*b*a`:

```
Vector = Vector * Vector * Vector.
```

This illustrates that each occurring `Vector` gets a different data pointer for its array. Hence we want the programmer to distinguish the vectors in their types. By adding a template integer to the `class Vector`

```
template <int vnum>  class Vector{...}
```

and declaring each vector by a different number

```
Vector<1> a; Vector<2> b; Vector<3> c;
```

this problem can easily be solved. However, no vector with the same template number may be declared twice because the aim is to support the compiler in recognizing multiple occurrences which would then fail. Let us apply this trick to our example, where the type information reads as

```
Vector<3> = Vector<1> * Vector<2> * Vector<1>.
```

Since now different vectors have different types the former problem remains. Equal vectors do not lead to the same data pointers. But we use this idea as a starting point for the following presentations.

6 FET by Expression-dependent Pointer Storage

To overcome this repeated creation of different pointers for the same data we implement the class `Pointer_Set` for storing only those pointers needed to evaluate an expression.

```
template <class C>
struct Pointer_Set{
  static double* d_[C::num+1];
  static void init(const X<C>& x){ x.cast().template Set<C>();}
  template<int pos>
  static void set(double* d){d_[pos] = d;}
  template<int pos>
  static double*& get () {return d_[pos];}
};
template<class C> double* Pointer_Set<C>::g_d[C::num+1];

template <class L, class R>
struct Times : public X<Times<L,R> >{
  const L& l;  const R& r;
  static const int num = _MAX_(L::num,R::num);
  Times(const L& l_, const R& r_) : l(l_), r(r_){}
  template <class C> double give(int i) const {
    return l.template give<C>(i)*r.template give<C>(i);}
  template <class C>
  void Set() const {l.template Set<C>();r.template Set<C>();}
};

template <int vnum>
class Vector : public X<Vector<vnum> > { ...
  static const int num = vnum;
  template<class A> void operator=(const X<A>& x){
    Pointer_Set<A>::init(x);
    for(int i = 0; i < N_ ; ++i)
      data[i] = x.cast().template give<A>(i);}
  template <class C>
```

```
   double give(int i) const {return Pointer_Set<C>::get<vnum>()[i];}
  template<class C>
   void Set() const {Pointer_Set<C>::set<vnum>(data);}
};
```

All members of `Pointer_Set` are static and the size of the static pointer array depends on the vectors in the expression. For the computation of this size we calculate the maximum of the occurring template integers in each operating class. In the assignment operator the according `Pointer_Set` array is filled with the pointers that are needed for the computation by invoking the `init(...)` function. Since the evaluation only takes its pointers out of this class by the call of `give(...)`, they are forced to be identical. But it has to be ensured that each vector has a different template integer. By this construct the compiler really works in the way we want it to and the performance for vector lengths larger than 50 is very close to the performance of handcrafted C code. However, for smaller sizes the problem of the extra storage in the operating and vector classes dominates and still causes performance lacks.

7 FET by Static Data Pointers

As a last improvement the data pointer of the vector classes is declared static, which is possible since each vector has a different template type.

```
template <class A> class X{};

template <class L, class R>
struct Times : public X<Times<L,R> > {
  static double give(int i){return L::give(i)*R::give(i);}
};
template <class L, class R>
inline Times<L,R> operator*(const X<L>& l, const X<R>& r){
  return Times<L,R>();}

template<int num>
class Vector : public X<Vector<num> >{ ...
  static double* data;
  const int N_;
 public:
  template<class A>
  void operator=(const X<A>& a) const {
    for(int i = 0; i < N_; ++i) data[i] = A::give(i);
  }
  static double give(int i) {return data[i];}
};
template<int num> double* Vector<num>::data;
```

By changing the function `give(...)` to static as well the evaluation in the operating classes can be called directly on the template types without storing any members. This implementation is very short and uses only the type of the template parse tree build from the expression. Without requiring extra storage of the pointers the performance of C code is reached because the static data pointers itself are not created anew for the evaluation. The static arrays are unique in the programming code and the compiler can thus work with this data pointers as with global variables.

However, some new technical problems arise from this implementation. Consider the expression template constructs for the multiplication of a vector and a double precision variable. This can easily be implemented using a class that stores the double precision number as astatic member set from within the constructor:

```
template <class A>
struct C : public X<C<A> >{
  C(double val_) {val = val_;}
  static double val;
  static double give(int i){return A::give(i)*val;}
};
template<class A> double C<A>::val;
```

The example `b = 2.*a * 3.*a` will reveal the problem. We follow the template creation and the call for component `i` in the assignment operator in the class `Vector`. Since the type of `2.*a` equals `3.*a`, the static member is set only once and the computation yields an incorrect result.

```
Vector<2> = 2 * a * 3 * a
          = C<Vector<1>>(2)*C<Vector<1>>(3)
          = Times<C<Vector<1>>,C<Vector<1>>>
Vector<2>::data[i] = Times<C<Vector<1>>,C<Vector<1>>>::give(i)
                   = C<Vector<1>>::give(i)*C<Vector<1>>::give(i)
                   = 2.*Vector<0>::data[i]*2.*Vector<0>::data[i]
```

A solution is to introduce an enumeration class to distinguish between the two double precision values, similar to the case of enumerated vectors. However, each double value in an expression must then be enumerated.

```
template <int i>  struct C : public X<C<i> >{ ... };
```

8 Comparison of the Different Implementations

In the following we compare the four implementations of ET discussed above with the corresponding handcrafted C code. For the test cases illustrated in the two figures we have implemented the component-wise addition and multiplication of vectors. All codes were compiled with the Gnu g++ compiler, version 3.3.3. The graphs show the time per iteration measured in seconds against the length of the used vectors, both axes with logarithmic scale. The first figure shows the computation

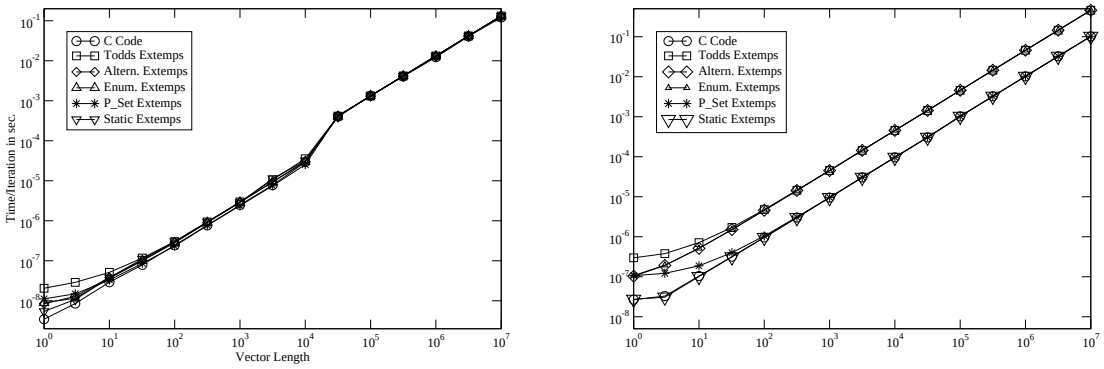

Figure 1: Performance Results on a Pentium 4 using the g++-3.3.3 compiler

of the expression `a = b + c * d` to present the improvements of FET even if vectors appear not repeatedly. For large vector lengths the performance of all implementations is the same. Only for small vector sizes they differ and the best ET approach to the C code is the static version. In the second figure we show a computation, where one vector occurs repeatedly:

```
a = a + a*a + a*a*a + a*a*a*a + a*a*a*a*a + a*a*a*a*a*a + a*a*a*a*a*a*a
```

Here the FET implementations perform up to ten times faster than the classical implementations. Only for small vector sizes one can see the difference between the external storage of the pointers and the static version. As mentioned above vector machines are more sensitive with aliasing. However, as tested so far the FET implementations can be vectorized by compilers running on those computers. But the usage of FET on vector computers is still in research and is not discussed here.

9 Conclusions

Since ET have been in use for about eight years their problems discussed by Bassetti, Davis and Quinlan led to a break in the euphoria. Recognizing the problems the classical implementations

suffer from we presented ideas to improve ET to work in the intended way. Of course, the organization effort of the enumerated variables is higher than simply using variables without template enumeration. A solution could be based on the automatic enumeration at compile time. As a simple solution the C++ macro `__LINE__` can be used. Then it has to be ensured that each vector is initialized in an different line. However, a different approach is possible. Since the static pointer version does not depend on the fact that the template type of the variables is an integer, a self-generating meta-template class type can probably be used. As the main result of this article we would point out the performance that can be reached by the presented FET implementations, particularly for small vectors. This technique can be used to reach performance as well as user-friendly mathematical interfaces in scientific codes.

References

[1] Bassetti, F, Davis, K, and Quinlan, D: Towards Fortran 77 Performance from Object-Oriented C++ Scientific Framework: HPC '98 April 5-9, 1998.

[2] Bassetti, F, Davis, K, and Quinlan, D: C++ Expression Templates Performance Issues in Scientific Computing. CRPC-TR97705-S, October 1997.

[3] Czarnecki, K, and Eisenecker, U: Generative Programming : Methods, Tools, and Applications. Addison-Wesley, Boston, 2000.

[4] Leibniz-Rechenzentrum München: The Hitachi SR8000-F1, System Description. http://www.lrz-muenchen.de/services/compute/hlrb/system-en

[5] High Performance Computing Center Stuttgart: The NEC SX-6 Cluster Documentation. http://www.hlrs.de/hw-access/platforms/sx6/user_doc

[6] Department of Computer Science 10, System Simulation, Erlangen: HPC Cluster http://www10.informatik.uni-erlangen.de/Cluster/hpc.shtml

[7] Los Alamos National Laboratories: PETE - Portable Expression Templates Engine. http://www.acl.lanl.gov/pete/html/introduction.html

[8] Pflaum, C: Expression Templates for Partial Differential Equations. Comput. Visual. Sci. **4**, 1–8, (2001).

[9] Los Alamos National Laboratories: POOMA: www.acl.lanl.gov/pooma

[10] Veldhuizen, T: Using C++ Template Metaprograms. *C++ Report* Vol. 7 No 4. (May 1995), pp. 36–43.

[11] Veldhuizen, T: Expression Templates. C++ Report **7** (5), 26–31 (1995).

[12] Veldhuizen, T: Will C++ be faster than Fortran? Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97).

[13] Veldhuizen, T: Blitz++. http://oonumerics.org/blitz/index.html

[14] Veldhuizen, T: Techniques for Scientific C++. Indiana University Computer Science Technical Report No 542, Version 0.4, August 2000.