Fast Expression Templates

Object-Oriented High Performance Computing

Jochen Härdtlein, Alexander Linke, and Christoph Pflaum

University of Erlangen, Department of Computer Science 10,
System Simulation Group, Cauerstr. 6, D-91058 Erlangen, Germany

Abstract. Expression templates (ET) can significantly reduce the implementation effort of mathematical software. For some compilers, especially for those of supercomputers, 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. The performance results of these so-called *Fast ET* are presented for the Hitachi SR8000 supercomputer and the NEC SX6, both with automatic vectorization and parallelization. Additionally we studied the combination of *Fast ET* and OpenMP on a high performance Opteron cluster.

1 Introduction

While object-oriented programming is being embraced in industry, its acceptance by the High Performing Computing Community is still very low, mainly because of supposed performance losses. Indeed, introducing abstract data types and operator overloading empowers the software engineer to forge interfaces very close to problem-specific notation, which can be understood by C++-compilers. However, there are no language features in C++ which are designed to inform the compiler about allowed transformations of mathematical expressions involving user-defined abstract data types. Therefore, though such user-defined mathematical expressions can be compiled by C++-compilers, they often perform very poorly. The first solution to overcome these performance problems was represented by the use of ET in C++, whose development we will summarize in the following.

Veldhuizen and Vandervoorde concurrently recognized the potential of templates in C++ and, 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. Unnecessary temporaries are avoided by performing some kind of expression-dependent inlining within a single loop. Therefore, less data

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 1055–1063, 2005.

is pumped through the memory hierarchy during the evaluation, thus enhancing code performance. For many systems the efficiency of ET implementations competes with their Fortran counterparts [12].

Soon, there was a rapid development of powerful mathematical packages based on ET; e.g., Blitz++ by Veldhuizen [13] and the Generative Matrix Computation Library (GMCL) described in [3]. Further important software frameworks are PETE [7] and POOMA [9], which represent two projects started at the Los Alamos National Laboratory. PETE is a tool for implementing ET for various applications. POOMA supports the implementation of mathematical algorithms for solving partial differential equations. Users of POOMA write sequential source code similar to FORTRAN 90 and benefit from getting automatic parallelization on various platforms just by using compiler switches. Some aims of the POOMA project are shared by the EXPDE project [8]. EXPDE eases the implementation of parallel 3D finite element codes by providing finite element and multigrid operators on arbitrarily shaped domains. Mathematical algorithms can be formulated with EXPDE in a language very close to the mathematical language. Furthermore, ET enable automatic parallelization.

Performance problems with ET were discovered for the first time by Bassetti, Davis, and Quinlan in 1997, see [1] and [2]. They studied the register spillage of ET codes and figured out the downsides of these implementations. Additionally to that paper one can observe thoroughgoing problems using ET on computers that enable automatic parallelization; e.g., vector machines or multiprocessor machines using OpenMP. Analyzing these problems, we developed, via the concept of enumerated variables, an implementation technique to assist the compiler overcoming these problems.

In the next section we will demonstrate how classical ET work and show the problems from which they suffer. After that we will present our implementation of Fast ET. In section 4 we will provide some examples comparing the described implementations of ET with C code computing the same problems. Section 5 concludes the paper.

2 Classical Expression Templates

For the sake of simplicity, we will focus on a minimal classical ET implementation for vectors with component-wise multiplication, as suggested in [14]. The mathematical operator `Times` is encapsulated by:

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

The expression class is then implemented as follows:

```
template<typename Left, typename Op, typename Right>
struct Expr {
  const Left &left_;  const Right &right_;
  Expr(const Left &t1, const Right &t2):left_(t1),right_(t2){}
```

```
  double Give(int i) const {
    return Op::apply(left_.Give(i), right_.Give(i));}
};
```

Here is a simple vector class:

```
class Vector {
  double * data_;
  const int N_;
 public:
  Vector(int N, double val);
  double Give(int i) const { return data_[i]; }

  template<typename Left, typename Op, typename Right>
  void operator= (const Expr<Left, Op, Right> &expr){
    for(int i=0; i < N_; ++i)
       data_[i] = expr.Give(i);
  }
};
```

Finally, the `operator*` is implemented as follows. The idea is to create a type that stores the information for the evaluation.

```
template<typename Left>
Expr<Left,Times, Vector>
 operator*(const Left &a,const Vector &b){
   return Expr<Left,Times, Vector>(a, b);
 }
```

We will now explain how ET work. Consider the line `c = a*b*a`. The type inference mechanism of the compiler can be represented as:

```
c = a*b*a;
  = Expr<Vector,Times,Vector>(a,b)*a;
  = Expr<Expr<Vector,Times,Vector>,Times,Vector>(
                          Expr<Vector,Times,Vector>(a,b),a);
  =: expr;
```

The call of `c.operator=` looks as follows:

```
c.operator=<Expr<Expr<Vector,Times,Vector>,Times,Vector> >(expr){
          for(int i=0; i < N_; ++i) data_[i] = expr.Give(i);}
```

Now `expr.Give(i)` can be expanded by inlining `Give(i)` from each node of the expression tree:

```
data_[i] = expr.Give(i);
  = Times::apply(expr.left_.Give(i),expr.right_.Give(i));
  = Times::apply(Times::apply(expr.left_.left_.Give(i),
                 expr.left_.right_.Give(i)),expr.right_.Give(i));
  = Times::apply(Times::apply(expr.left_.left_.data_[i],
                expr.left_.right_.data_[i]),expr.right_.data_[i]);
  = expr.left_.left_.data_[i] * expr.left_.right_.data_[i]
                                          * expr.right_.data_[i];
```

If `operator=` is inlined and the aliasing for the pointers works, a C++ compiler can, in principle, optimize the expression such that it correspond to the following code fragment:

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

This is just the code a C programmer would naturally write. Indeed, to reach this stage a compiler must have good optimization facilities. Since not all compilers are capable of C++-specific optimization facilities, the resulting programs based on classical ET can suffer from a variety of performance problems. Although component-wise multiplication of three vectors is a very trivial application of ET, the intermediate C++ code inlined by ET is quite complex. More difficult applications such as ET for discretized differential operators in 3D yield much more complex intermediate code.

The main problems of ET are discussed in [1] and [2]. For a better understanding of the performance lacks we focus again on the example presented above. The pointers `expr.left_.left_.data_` and `expr.right_.data_` represent the same data array. The aliasing concept of the C++ compiler must be able to resolve this to the pointers they denote. Otherwise there will be two different pointers that represent the same data. During the evaluation loads are issued, even if the data has already been loaded into the CPU core. This obviously causes performance lacks. Moreover, as aliasing is even more important on vector machines the solution of this problem can significantly reduce the performance problems in object-oriented high performance computing.

3 Fast Expression Templates

After this short discussion of classical ET and their problems, we present an alternative approach based the concept of enumerated variables. The idea is to equip the `Vector` class with an additional integer parameter which is unique for each vector variable. The aim is to communicate to the compiler if the variables differ or not. Our second idea is to derive an expression object from an expression whose information is completely accessible at compile time. This allows more intelligent inlining and guarantees enhanced C++ performance.

```
template <class A, class B>
struct Times : public Expr<Times<A,B> > {
  static inline double Give(int i){return A::Give(i)*B::Give(i);}
};
template <class A, class B>
inline Times<A,B> operator*(const Expr<A>& a, const Expr<B>& b){
  return Times<A,B>();
}
```

Moreover, the `Give` function for the `Vector` class has to be static as well. However, the vector components and even the vector size is still unknown at compile time. We solve this problem by introducing a static data pointer. This is

reasonable because different variables have different types. Consider the following implementation of the `Vector` class:

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

Last but not least, the excerpt from the main program looks as follows:

```
Vector<0> a(N, 1.);  Vector<1> b(N, 3.);  Vector<2> c(N, 0.);
c = a*b*a;
```

We will now explain how this new approach differs from classical ET. To this end, we focus again on the expression `c=a*b*a`, whose expression object has the following type:

```
Times<Times<Vector<0>,Vector<1>>,Vector<0>>
```

Since this expression type contains the complete information about the expression `c=a*b*a`, more efficient inlining is possible. The expression is evaluated as:

```
c.operator=
  Times<Times<Vector<0>,Vector<1>>,Vector<0>>(expr){
   for(int i = 0; i < N; ++i)
     c.data_[i] = Times<Times<Vector<0>,Vector<1>>,Vector<0>>::Give(i);
}
```

Now we demonstrate how the static `Give(i)` function can be inlined:

```
Vector<2>::data_[i] =
  = Times<Times<Vector<0>,Vector<1>>,Vector<0>>::Give(i);
  = Times<Vector<0>,Vector<1>>::Give(i) * Vector<0>::Give(i);
  = Vector<0>::Give(i) * Vector<1>::Give(i) * Vector<0>::Give(i);
  = Vector<0>::data_[i]*Vector<1>::data_[i]*Vector<0>::data_[i];
```

This inlined source code corresponds to the code a programmer would naively write. In comparison to classical ET where the pointers `expr.right_.data_` and `expr.left_.left_.data_` represent the same data array, here the pointer `Vector<0>::data_` occurs twice. Hence, Fast ET support the compiler in aliasing identical data arrays.

Working only with the type of an expression means using only that information which is really needed to build the evaluation. Enumerating is just the base

for reaching this goal. In general the enumeration concept with static data joins two ideas of programming. On one hand, it introduces the use of global variables that are easier to optimize for compilers, because there are less aliasing conflicts than in pure object-oriented codes. On the other hand, enumerated variables enable data encapsulation in the object-oriented sense. And joining all in Fast ET we apply well performing operator overloading.

4 Performance Results

After the discussions of classical ET (CET) and Fast ET (FET) we present some performance results. The two implementations are compared with a C code that uses no ET (NET) but computes the same problem. In addition to the multiplication we implemented component-wise sum of vectors in the same manner. As we are interested in the effort of ET in high performance computing we tested these implementations to this aspects. We analyzed two types of expressions, at first the vector triad $a = b + c * d$ that does not suffer from the aliasing problem because each vector only occurs once. The second expression is $a = \sum_{i=1}^{7} a^i$. Obviously the efficiency of its implementation is highly dependent on the compiler's capabilities to detect array aliasing.

The results are presented for three platforms. At first the NEC SX-6/48M6 at the HLRS in Stuttgart, which is a shared memory vector system. We started the computation on one node with automatic vectorization. The sxc++ compiler enables the vectorization. The second platform is the Hitachi SR8000 supercomputer at the LRZ Munich. A single node is equipped with eight RISC processors and each processor can perform vector operations with floating point numbers, see [4]. On the Hitachi SR8000 we used the optimizing C++ compiler sCC by Hitachi, because it is the only C++ compiler which can vectorize C++ programs on this platform. At last we demonstrate the performance tests on a

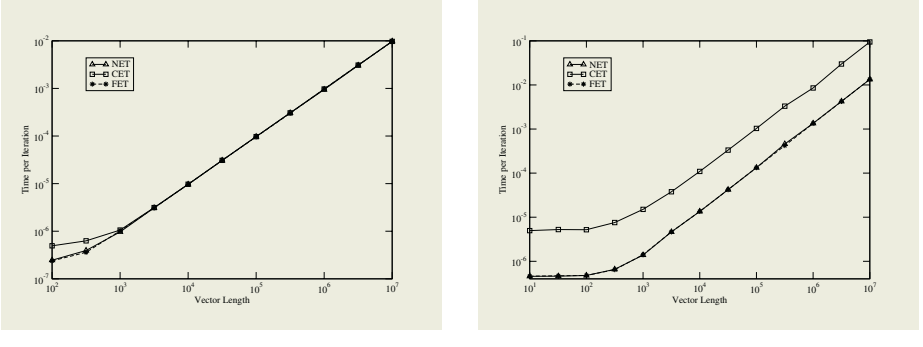

Fig. 1. Performance results on the Nec SX-6/48M6. On the left side the graph for the vector triad where only small vector lengths show the improvement by FET. The right figure presents the performance of the expression $a = \sum_{i=1}^{7} a^i$. FET run more than ten times faster than CET even for large vectors

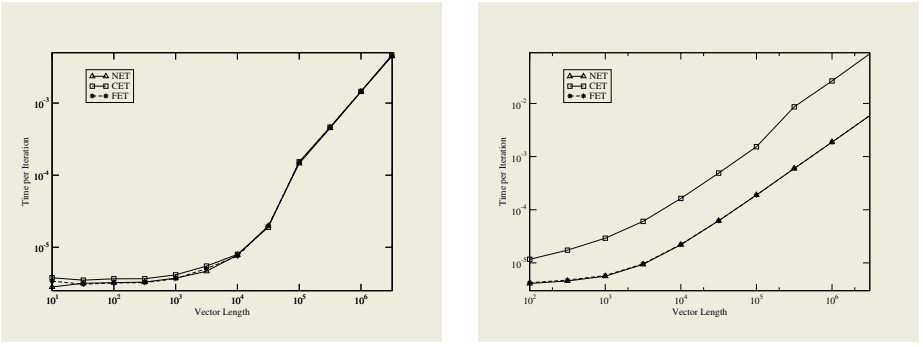

Fig. 2. $a = b + c * d$ and $a = \sum_{i=1}^{7} a^i$ on the Hitachi SR8000 supercomputer. The first example points out a small improvement by FET in comparison to CET mainly for small vector sizes. The right figure demonstrates the performance enhancement by the FET implementation

AMD Opteron Cluster with dual-nodes and quad-nodes. On this platform the implementations are translated by the Intel C++ compiler, version 8.1, and the evaluation is started on a dual node.

The results of the two examples are presented below. We measured the time per iteration over the vector length. At first, the vector triad does not suffer from lacks in aliasing. Only for small vector sizes the FET work faster than CET. This is reasonable because, unlike CET, the FET implementations do not store the expression members in the operating classes. However, this effect is compiler-dependent; e.g., the Intel C++ compiler optimizes all the nested function calls through the classical ET constructs in the expected way (see Fig. 3). Hence, there are also no differences in the performance for small vector lengths. The second example shows the real improvements by FET. In the majority of cases, they

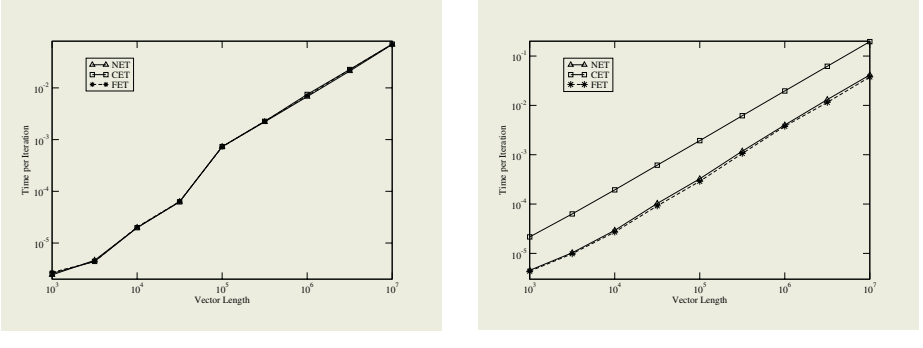

Fig. 3. The Opteron Cluster using Open MP and the Intel C++ compiler does not arise performance differences for the vector triad. This is due to the good optimizing facilities of the Intel compiler. However, the big expression in the right graph confirms the improvements resulting from the application of FET

work as well as the C code does. Only some small oscillations can be observed. The CET implementation suffers from the lacks in aliasing. The compiler does not recognize the repeated occurrence of the data pointer of the vector a. Hence, during the evaluation loads are issued even though the data has already been loaded into the CPU core. already in cache. This causes the performance lacks that slow down the implementation more than ten times compared to the other ones.

5 Conclusions and Perspectives

While ET have been in use for about eight years now, their problems discussed by Bassetti, Davis, and Quinlan led to a break in the euphoria. Recognizing the lacks of classical implementations, we introduced a concept to avoid these problems. Our new implementation of ET supports the compiler in solving the aliasing problems. Of course, there is an extra effort in organizing FET. Every template integer of an enumerated variable has to be unique within the whole program code, because they work, in principle, like global variables. Using FET in a library yields the enumeration of the variables to the user. A simple solution is the use of the `__LINE__` macro and defining every vector in a different line.

The performance that could be reached by using FET instead of the classical implementations is near to the C code performance. Hence, FET potentiates the ideas of a user-friendly interface and a well performing code.

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.