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

Lehrstuhl für Informatik 10 (Systemsimulation)

Blocking Techniques with Fast Expression Templates

Jochen Härdtlein, Alexander Linke and Christoph Pflaum

Lehrstuhlbericht 06-8

Blocking Techniques with Fast Expression Templates

Jochen Härdtlein, Alexander Linke and Christoph Pflaum

Abstract

The use of Expression Templates (ET) can significantly reduce the implementation effort for scientific codes. Thereby, programmers are able to build user-friendly, problem-specific interfaces, which perform much better than implementations using traditional operator overloading. However, ET still suffer from performance lacks, caused by aliasing problems, and poor blocking.

The aliasing problems became solved by an enhanced implementation technique, so-called Fast Expression Templates (FET), which result a performance comparable to their C counterparts. The second issue, poor blocking, is addressed in this article. Since FET implementations work with template types through the evaluation, thus, this technique offers ways to realize loop manipulation techniques.

1 Introduction

The C++ programming language enables programmers to build libraries, that provide user-friendly interfaces, by exerting operator overloading and object-oriented programming techniques. Hence, users of application ranges can easily use those libraries and, thereby, significantly reduce the implementation effort. Focusing on the solution of partial differential equations (PDE), those C++-libraries already exist and they are used in industrial applications; e.g., the ExPDE written by Pflaum, see [Pfl01].

The ability to implement user-friendly libraries is one of the advantages of the C++ programming language. However, those problem-specific, object-oriented codes often suffer from several performance lacks. As the numerical solution of PDEs means dealing with big data arrays and permantently increasing problem sizes, performance turns out to be more important than an easy usage. But there are approaches to realize both, efficiency and user-friendly interfaces.

Expression Templates (ET) were the first significant improvements to traditional operator overloading, concurrently introduced by Veldhuizen and Vandervoorde in 1995, see [Vel95a] and [Vel95b]. Traditional operator overloading makes use of temporary objects at the evaluation, causing a very memory-intensive data-transfer. By using ET implementations the evaluation is firstly started via the assignment operator. A nested template type covering the expression parts is built by the operators, instead of immediately calculating the result. During the setup of the template expression, no memory-intensive data-transfer is needed, and by inlining the evaluation is computed in one single loop. Therefore, ET perform much better than implementations using traditional operator overloading. However, some problems still remain, see [BDQ98] and [BDQ97].

There are two main lacks arising from ET, discussed in [BDQ97]. First, the nested template constructs prevent C++ compilers to apply the designated aliasing concepts. Thus, ET implementations still cause performance lacks, while computing expressions, where variables occur repeatedly. As a well working aliasing concept is more important on high performance machines; e.g. vector machines; such applications of ET implementations perform still worse. Second, the operator overloading and, thereby, ET separates the evaluation loop from enclosing iteration loops. This separator prevents the compiler as well as the programmer to apply loop manipulation techniques. For instance, the performance of many grid-based algorithms can be improved by blocking techniques. Thus, this separation serves as reason for critics to avoid ET implementations. Additionally, the implementation of ET is quite complex, and the programmer needs some experiences, in order to implement them correctly. Otherwise the program could slow down to the performance of the traditional operator overloading.

As introduced in [HLP05] the performance of ET and their implementation effort can be improved. A summery of the Fast Expression Templates technique is explained in the following section. Starting from this technique, we present the implementation manners to enable blocking.

2 Fast Expression Templates (FET)

2.1 Implementation Policies

This section provides an introduction to the FET programming technique. We demonstrate the implementation technique using the component-wise computations of a vector class. This ought to be an introduction, even if the reader has not yet worked with ET. But for those programmers, who have experiences in programming ET, we summerize the main differences and enhancements of FET. Considering a classical ET implementation, the FET counterparts differ in four main points:

1. template enumeration of all variables,
2. inheriting all working classes from the wrapper class,
3. declaring all data and accessing functions as static, and
4. calling the evaluation on the corresponding template types .

In detail, the structure of a FET implementation is divided into three parts. At first, we focus on the common interface. This so-called wrapper is an empty class, covering the working classes as template type, and all working classes inherit from this wrapper.

```
template <class A> struct Expr { };
```

Second, we adapt the basic vector class. The class `Vector` provides the storage of the vector data and the methods to manipulate a vector. In order to access the data by static functions, the data pointer has to be declared as static. Therefore, the vectors are enumerated by a template integer, that has to be unique for every occurring vector variable, since varying static data pointers need different data types. Otherwise, FET would work with the same data for different vectors. For an easy description of the `Vector` class consider the following code fragment:

```
template <int num>
 class Vector : public Expr<Vector<int> > {
   static double* data_;
   int N_;
  public:
   Vector(int N, double w) : N_(N) {
     data_ = new double*[N_];
     for(int i=0; i<N_; ++i) data_[i] = w;
   } // Other constructors, destructor ...

   static inline double give (int i) {return data_[i];}

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

At last, we present the component-wise sum of vectors. The multiplication is coded in the same manner. The informations that are necessary for the evaluation of the expression is encapsulated in the template types of both summation parts of the class `Add`.The class `Add` inherits from the wrapper class, as well, so as to achieve the common interface `Expr`.

```
template <class A, class B>
 struct Add : public Expr<Add<A,B> > {
   static inline double give(int i) {return A::give(i) + B::give(i);}
 };
```

Finally, the overloaded `operator +`, which returns the corresponding addition expression object.

```
template <class A, class B>
 inline Add<A,B> operator+(const Expr<A>& a, const Expr<B>& b){
   return Add<A,B>();
 }
```

2.2 Usage and Performance

As only difference in comparison to classical ET, all variables containing data have to be enumerated. Consider three vectors `a`, `b`, `c`. Since every vector must have its own unique template integer, the variables are initialized with different template numbers:

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

After initializing the vectors, any expression of vectors can be written very easily; e.g., the component-wise addition and multiplication of vectors:

```
c = b + a + b * a ;
```

The calls of the operators `+` and `*` result an expression tree, that is stored as nested template construct. The corresponding template type of the right hand side expression looks like:

```
Add < Vector<2>, Add < Vector<1>, Mult < Vector<2>, Vector<1> > > >
```

As the static data pointers are unique and handled by the compiler similar to global pointers, inlining and optimization can be applied. The evaluation of the whole expression is started in one single loop at the assignment operator.

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

Thus, FET implementations perform much better than their classical ET counterparts and reach the efficiency of handcrafted C-code. As validation we present performance results on a Pentium 4, using the Intel C++-compiler, version 8.1, and a NEC SX 6 vector machine, see [Stu03]. We computed Poisson's problem in 2D, applying a vectorizable Jacobian solver, and testing the three previous implementation policies, as well. Hence, the performance of FET is again equals the C counterpart and is up to five times faster than the classical ET implementation. Both axis are provided with logarithmic scale.

For more detailed information and performance results on other platforms we refer to [HLP05]. By using the FET implementation above, an unique enumeration over several files needs some effort. Thus, we provide a possibility to support the user. Since the variables have to differ in their type, we extend the template of the class `Vector` by a `char*` template. Additionally, a macro for declaring variables is provided:

```
#define VECTOR Vector<__LINE__,__FILE__>
```

Thus, the user has just to ensure, that every variable is declared in an own line. Variables declared in different files result different types.

3 Blocking and FET

After introducing FET we concentrate on loop manipulation issues. As FET enable a compiler to realize the evaluation in a single loop, any outer loop is not yet involved in the optimization. An outer loop; e.g. the iteration loop of an Jacobian solver; is separated from the inner loop by the nested template parameters and the operator overloading. The inner loop is evaluated before the outer loop is updated. Since the loop iteration data; i.e. beginning, end, and stride; is not fused within the loop inside the assignment operator, we store the loop data, instead of evaluating the loop. As storage consider a struct `Evaluate`, containing two template parameters:

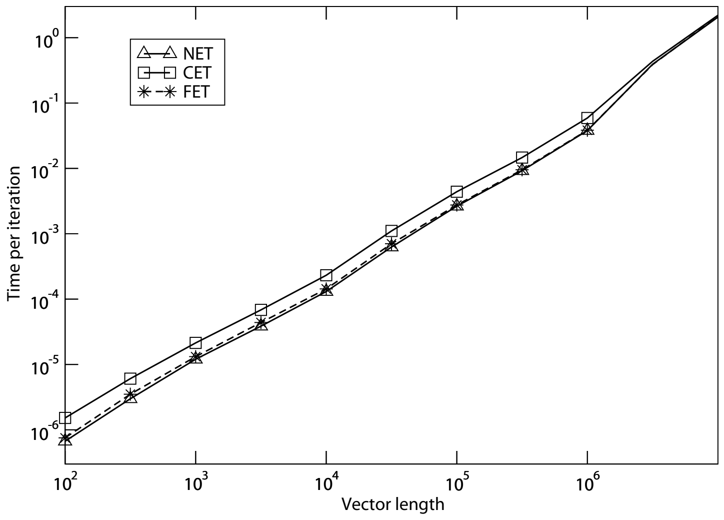

Figure 1: Performance results on a Pentium 4, using the Intel C++-compiler, version 8.1. The graph shows a Gauss-Seidel solver applied on Poisson's problem in 2D, and using the three previous implementation policies, as well. Hence, the performance of FET is again equals the C counterpart and is up to two times faster than the classical ET implementation. Both axis are provided with logarithmic scale.

```
 template <class A, class B>
   struct Evaluate { };
```

The main idea is to delay the evaluation to the instantiation of a `For` object that simulates an outer loop. The new implementation of the assignment operator in the vector class does not compute the result, but returns an `Evaluate` object containing the result vector and the expression as template type.

```
  template <class A>
   inline Evaluate<Vector<num>,A>
    operator = (const Expr<A>& a){
      return Evaluate<Vector<num>,A>();
    }
```

Before the specific loop class is introduced, we present a parent class providing all iteration parameters as static variables. The `Loops` class is declared as friend to `Vector`, hence, it has access to the private variables of the basic classes. The inherit is realized at compile-time by the Barton-Nackman-Trick, see [Vel00]. For the sake of simplicity, the following code fragment omits the allocation of the static data. In the parenthesis operator the inner loop size is set and the suitable iteration version of the child is called:

```
template <class A>
  class Loops{
   protected:
    static int o_start, o_size, o_stride, i_start, i_size, i_stride;
   public:
    Loops(int b, int e, int s) {
      o_start = b; o_size = e; o_stride = s;
      i_start = 0; i_stride = 1;
    }

    template <class B, class C>
    inline void operator () (const Evaluate<B,C>& ev){
```

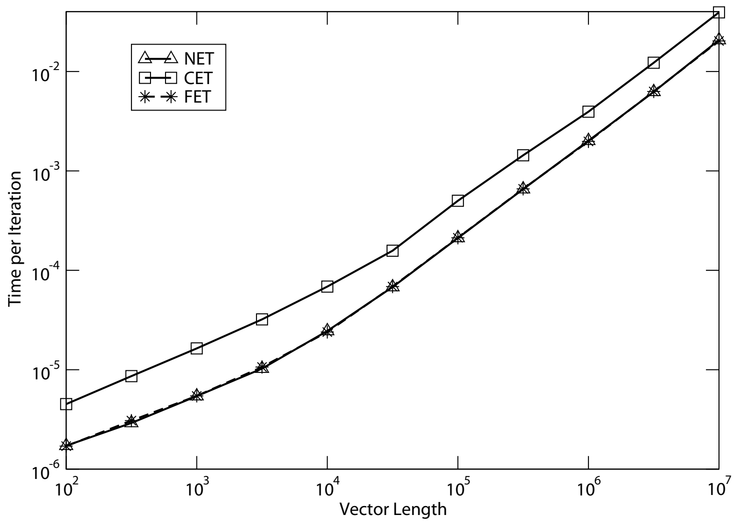

Figure 2: Performance results on a NEC SX 6 vector machine. We computed Poisson's problem in 2D, applying a vectorizable Jacobian solver, and testing the three previous implementation policies, as well. Hence, the performance of FET is again equals the C counterpart and is up to five times faster than the classical ET implementation. Both axis are provided with logarithmic scale.

```
      i_size = B::size();
      A::template iteration<B,C>();
    }
  };
  // initialization of the static integers ...
```

The `For` class has to process the iteration of both loops, using the data stored in `Loops<For>`. These values are set by the constructor of the class `For`. To simulate the loop-like call, the inherited parenthesis operator invokes the static function `iteration`, that starts the evaluation:

```
struct For : public Loops<For> {
  For(int b, int e, int s=1) : Loops<For>(b,e,s) {}

  template <class A, class B>
  static inline void iteration(){
    int k = 0, i = 0;
    for(k = o_start; k < o_size; k+=o_stride)
      for(i = i_start; i < i_size; i+=i_stride)
        A::Set(i) = B::Give(i);
  }
};
```

At least, we explain the usage of this loop in application. The syntax ought to be similar to the `for` loop in C++. The `For` class constructor is initialized with the beginning, size, and stride of the outer loop. Additionally, the constructor is followed by the expression inside the parenthesis:

```
For(0,itermax,1) (
                   a = b + c * d
                                   );
```

This implementation of `iteration` has no effect to the performance yet, since the C++-compiler is not able to realize a loop interchange or splitting on its own. Hence, we assist the compiler with blocking techniques. The trick is to split the inner loop in parts that fit into cache, in order to reduce the loads. While the cache size for a platform is known, the amount of data that fits into cache is expression-dependent. More precisely, it depends on the number of occurring variables.

In order to count the different variables in an expression we use type-lists, see [Ale01]. Type-lists are nested template constructs of types. Since FET provide every vector with its own type, we build a type-list of the vectors occurring in the expression, erase duplicates, and count the number of variables at compile-time. Then, the size and number of the blocks for the splitting of the loops can be computed. The compile-time counting of the variables is just hinted by the `Length(A,B)` since the complete implementation covers 80 lines and would blow the paper:

```
struct Loop : public Loops<Loop> {
  Loop(int b, int e, int s=1) : Loops<Loop>(b,e,s) {}

  template <class A, class B>
  static inline void iteration(){
    int k = 0, i = 0, j = 0, block_start = i_start;
    const int vars  = Length(A,B),
              num   = (vars * i_size)/_CACHE_SIZE_ + 1,
              block = i_size/num;
    for(k = 0; k < num; ++k){
      for(i = o_start ; i < o_size; i+=o_stride)
        for(j = block_start; j < block_start + block; j+=i_stride)
         A::Set(j) = B::Give(j);
      block_start += block;
    }
    for(i = o_start ; i < o_size; i+=o_stride)
       for(j = block_start;j <i_size; j+=i_stride)
         A::Set(j) = B::Give(j);
  }
};
```

Naturally, this `Loop` class is used in the same manner as the `For` class presented before.

Blocking techniques can increase the performance of loops; e.g. see [Kow04]. Figure 3 figure confirms the success of the last loop implementation. The results were computed on a Pentium 4, using the Intel C++-compiler, version 8.1. We compared handcrafted C-code, C-code with blocking techniques and the FET counterpart by computing Poission's problem in 2D using a Gauss-Seidel solver.

4 Conclusions and Further Work

Expression Templates are in use since 1996. While the performance lacks of ET, concerning the aliasing problems, were solved in [HLP05], the second point of criticism, poor blocking, is started to get loosen. The evaluation loop inside the ET constructs can be merged with outer loops, in order to reach better performance by applying loop manipulation techniques. Users can implement oneself and easily swap varying loop versions without wide chances in their applications.

The disadvantage of FET is the fact, that the vectors must be enumerated by the user. This problem is lowered by the template extension for the `Vector` class, as presented before. However, we intend to implement an automatic enumeration of the variables, so that the user has not to care about this issue anymore.

Since the blocking and variable counting as shown above is a simple way to reach better performance in scientific codes, this concept invites the user to think of suitable loops himself. Hence, more complex and problem-specific iteration methods can be covered by those user-defined `Loop` classes. Since the FET technique can be applied on high performance platforms, as well, the efficiency of array-based codes can be increased on such platforms in the same manner.

Summarized, this paper presented a way to reach an user-friendly and high-performance library for solving array-based problems. Since FET perform very well for small vector sizes, too, this techniques can be used for a large field of applications; e.g. the computation of local stiffness matrices, see [HP05].

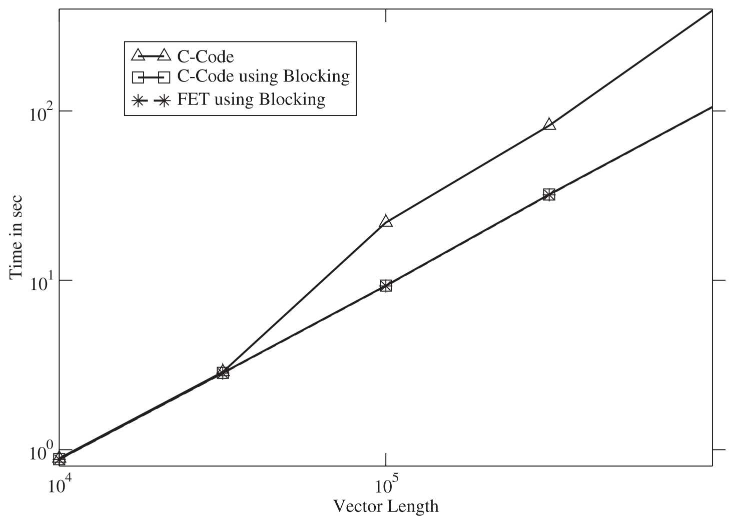

Figure 3: The graph figures the performance results on a Pentium 4, using the Intel C++ compiler, version 8.1. We computed Poisson's problem in 2D using a Gauss-Seidel solver via hand-crafted C-Code without blocking, C-Code applying blocking techniques, and the FET with blocking. The blocking performance of the FET implementations equals the efficiency of the hand-crafted C-code with blocking. Every implementation was stopped when reaching the same discretization error. Thus, the y-axis is scaled in logarithmic time.

References

[Ale01] A. Alexandrescu. *Modern C++ Design, Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[BDQ97] F Basetti, K Davis, and D Quinlan. C++ expression templates performance issues in scientific computing, Oct 1997. CRPC-TR97705-S.

[BDQ98] F Basetti, K Davis, and D Quinlan. Toward fortran 77 performance from object-oriented c++ scientific framework, Apr 1998. HPC '98.

[HLP05] J. Härdtlein, A. Linke, and C. Pflaum. Fast expression templates. In V.S. Suneram, G.D.v. Albada, P.M.A. Sloot, and J.J. Dongarra, editors, *Computational Science - ICCS 2005*, volume 3515 of *LNCS*, pages 1055 – 1063. Springer, May 2005. ISBN-10 3-540-26043-9, ISBN-13 978-3-540-26043-1, ISSN 03-2-9743.

[HP05] J. Härdtlein and C. Pflaum. Efficient and user-friendly computation of local stiffness matrices. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *18th Symposium Simulationstechnique ASIM 2005 Proceedings*, volume 15 of *Frontiers in Simulation*, pages 748–753. ASIM, SCS Publishing House, Sep 2005. ISBN 3-936150-41.

[Kow04] M. Kowarschik. *Data Locality Optimizations for Iterative Numerical Algorithms and Cellular Automata on Hierarchical Memory Architectures.* PhD thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, Universität Erlangen-Nürnberg, July 2004. SCS Publishing House, ISBN 3-936150-39-7.

[Pfl01] C Pflaum. Expression templates for partial differential equations, 2001. Comput Visual Sci **4**, 1–8.

[Stu03] High Performance Computing Center Stuttgart. The nec sx-6 cluster documentation, 2003. http://www.hlrs.de/hw-access/platforms/sx6/user_doc.

[Vel95a] T Veldhuizen. Expression templates, 1995. C++ Report **7** (5), 26–31.

[Vel95b] T Veldhuizen. Using c++ template metaprograms, May 1995. *C++ Report* Vol. 7 No 4., pp. 36–43.

[Vel00] T Veldhuizen. Techniques for scientific c++, Aug 2000. Indiana University Computer Science Technical Report No 542, Version 0.4.