SAND2012-0069 C

SAND2012-0069C

Efficient Expression Templates for Operator Overloading-based Automatic Differentiation

Eric Phipps and Roger Pawlowski

Abstract Expression templates are a well-known set of techniques for improving the efficiency of operator overloading-based forward mode automatic differentiation schemes in the C++ programming language by translating the differentiation from individual operators to whole expressions. However standard expression template approaches result in a large amount of duplicate computation, particularly for large expression trees, degrading their performance. In this paper we describe several techniques for improving the efficiency of expression templates and their implementation in the automatic differentiation package Sacado [15, 16]. We demonstrate their improved efficiency through test functions as well as their application to differentiation of a large-scale fluid dynamics simulation code.

Key words: Forward mode, operator overloading, expression templates, C++

1 Introduction

Automatic differentiation (AD) techniques for compiled languages such as C++ and Fortran fall generally into two basic categories: source transformation and operator overloading. Source transformation involves a preprocessor that reads and parses the code to be differentiated, applies differentiation rules to this code, and generates

Eric Phipps
Sandia National Laboratories†, Optimization and Uncertainty Quantification Department, Albuquerque, NM, USA, etphipp@sandia.gov

Roger Pawlowski
Sandia National Laboratories†, Multiphysics Simulation Technologies Department, Albuquerque, NM, USA rppawlo@sandia.gov

†Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

new source code for the resulting derivative calculation that can be compiled along with the rest of the undifferentiated source code. This approach allows for generation of efficient derivative code as it has a global view of the calculation, and thus is quite popular, particularly for simpler languages such as Fortran and C. However source transformation for C++ is challenging due to the complexity of the language. An alternative approach for C++ (and many other languages) is operator overloading. Here new derived types storing derivative values and corresponding overloaded operators are created so that when the fundamental scalar type in the calculation (float or double) is replaced by these new types and evaluated, the relevant derivatives are computed as a side-effect. This approach is attractive in that it uses native features of the language, making operator overloading-based AD tools simple to implement and use. There are two basic challenges for operator overloading schemes however: run-time efficiency and facilitating the necessary type change from the floating point type to AD types. For forward-mode AD, expression templates can be used to partially address the first of these challenges. However achieving the full performance benefits of expression templates is challenging and is the subject of this paper. For the second challenge, we advocate a templating-based approach which has been extensively described elsewhere[4, 13, 14, 15].

This paper is organized as follows. We first describe standard expression template techniques and their application to forward mode automatic differentiation in Sect. 2. For concreteness, we describe the simple implementation of these techniques in the AD package Sacado [15, 16]. Then in Sect. 3 we describe two techniques for improving the performance of expression templates: expression-level reverse mode and value caching. We demonstrate significantly improved performance for these techniques, particularly for large expressions, by applying them to two test functions. We then briefly describe applying all of these techniques to a large-scale fluid dynamics simulation in Sect. 4, again demonstrating improved performance on a real-world application. We then close with several concluding remarks in Sect. 5.

2 Expression Templates for Forward Mode Automatic Differentation

As described above, operator overloading-based AD schemes work by first creating a new derived type and corresponding overloaded operators. While operator overloading can be used for any AD mode, and there are many ways of implementing the overloaded operators for any given AD mode, we will restrict this discussion to tapeless implementations of the first-order vector forward mode. Here the AD type typically contains a floating point value to represent the value of an intermediate variable, and an array to store the derivatives of that intermediate variable with respect to the independent variables (see [11] for an introduction to basic AD implementations). The implementation of each overloaded operator then involves calculation of the value of that operation from the values of the arguments and stored

in the value of the result, and a loop over the derivative components using the corresponding derivative rule from basic differential calculus.

There are two basic problems with this approach. First, each intermediate operation within an expression requires creation of at least one temporary object, and creating and destroying this object adds significant run-time overhead. Second, the AD implementation is limited to differentiating one operation at a time, each involving a loop over derivative components. Together these problems often result in inefficient derivative code. Expression templates are a technique that was invented to address these issues. They were first used in the Fad package [2, 3], and later incorporated into Sacado. Here the AD type is fundamentally the same, however the operators return an object encoding the type of operation and a handle to their arguments instead of directly evaluating the derivative. As each term in the expression is evaluated, a tree is created encoding the structure of the whole expression. Then the assignment operator for the AD type loops through this tree recursively applying the chain rule. An implementation of these ideas for the $\times$ operator is shown below.

Listing 1 Partial expression template-based operator overloading implementation.

```
// Expression template-based Forward AD type
template <typename T> class Expr {};
class ETFadTag {};
class ETFad : public Expr<ETFadTag> {
  double val_;               // value
  std::vector<double> dx_;   // derivatives
public:
  explicit ETFad(int N) : val_(0), dx_(N) {} // Constructor
  int size() const { return dx_.size(); }
  double  val() const { return val_; } // Return value
  double& val()       { return val_; } // Return value
  double  dx(int i) const { return dx_[i]; } // Return derivative
  double& dx(int i)       { return dx_[i]; } // Return derivative

  // Expression template assignment operator
  template <typename T> ETFad& operator=(const Expr<T>& x)  {
    val_ = x.val();
    dx_.resize(x_.size());
    for (int i=0; i<x.size(); i++)
      dx_[i] = x.dx(i);
  }
};

// Specialization of Expr to multiplication
template <typename ExprT1, typename ExprT2>  class MultTag {};
template <typename T1, typename T2>
class Expr< MultTag< Expr<T1>, Expr<T2> > > {
  const Expr<T1>& a_;
  const Expr<T2>& b_;
public:
  Expr(const Expr<T1>& a, const Expr<T2>& b) : a_(a), b_(b) {}
  int size() const { return a.size(); }
  double val() const { return a.val() * b.val(); }
  double dx(int i) const { return a.val()*b.dx(i)+a.dx(i)*b.val(); }
```

```
};

// Expression template implementation of a*b
template <typename T1, typename T2>
Expr< MultTag< Expr<T1>, Expr<T2> > >
operator * (const Expr<T1>& a, const Expr<T2>& b) {
  return Expr< MultTag< Expr<T1>, Expr<T2> > >(a,b);
}
```

Each operator returns a simple expression object that stores just references to its arguments with the kind of operation encoded in the type of the expression. Since these expression objects just contain references, a good optimizing compiler can often eliminate them all together and generate code that is functionally equivalent to that shown below when applied to $d = a \times b \times c$.

Listing 2 Equivalent derivative code resulting from differentiation of $a \times b \times c$.

```
d.val() = a.val()*b.val()*c.val();
for (int i=0; i<d.size(); i++)
  d.dx(i) = (a.val()*b.val())*c.dx(i) +
    (a.val()*b.dx(i)+a.dx(i)*b.val())*c.val();
```

Thus all of the intermediate temporary AD objects have been removed and the loops have been fused into a single loop over the derivative components for d. This often removes much of the overhead associated with a simple operator overloading approach. We note that constants and passive variables introduce additional complexity into the implementation which is not shown or discussed here.

3 Improving Performance of Expression Templates

While the expression template approach can significantly reduce the overhead associated with operator overloading, there is still room for improvement in reducing the cost of the differentiation. Careful examination of the functionally equivalent derivative code in Listing 2 reveals a basic problem: the calculation of the value portion of intermediate terms in the expression can be repeated multiple times. In this case, the value `a.val()*b.val()` is recomputed N times where N is the number of derivative components. This is particularly troublesome for large expressions involving many terms or expressions involving transcendental functions whose values are expensive to compute.

We have investigated overcoming this problem by caching the value of each intermediate operation in the expression in the expression objects themselves. The small modifications to the multiplication expression template from Listing 1 are shown below where the `cache()` method in this case stores the result of `a.val()*b.val()`. This cached value is then used in any subsequent calls to `val()`. The top-level expression-template assignment operator is then modified to call `cache()` before any calls to `val()` or `dx()`. Other nonlinear operations may also cache partial derivatives with respect to their arguments for use in subsequent `dx()` calls. This approach eliminates the

duplicate computation of intermediate values, at the expense of more complicated expression objects that the compiler may not be able to optimize away. Nonetheless we have found this approach more efficient for recent compilers that support aggressive C++ optimization.

Listing 3 Modifications for caching expression template-based operator overloading.

```
template <typename T1, typename T2>
class Expr< MultTag< Expr<T1>, Expr<T2> > > {
  mutable double val_;
public:
  void cache() const {
    a.cache(); b.cache(); val_ = a.val() * b.val();
  }
  double val() const { return val_; }
};
```

A second technique that can be used to generally improve the performance of forward-mode AD is expression-level reverse mode [7]. This results from the recognition that while derivatives are generally being propagated forward through the calculation, any given expression likely has multiple inputs and only one output. Thus it should be more efficient to compute the derivative of the expression outputs with respect to its inputs using reverse-mode AD and then combine those derivatives with the derivatives of the inputs with respect to the independent variables using the chain rule. While this technique is common in source transformation tools such as ADIFOR [6], we are unaware of any use in operator overloading tools due to their limited view of the code being differentiated. Expression templates however do provide all of the necessary information for this technique. The challenge is implementing the technique in such a way as to allow the compiler to generate efficient derivative code.

We have implemented expression-level reverse mode within the forward mode classes in our tool Sacado using template meta-programming techniques [1] similar to those found in the Boost MPL library [9]. Referring to Listing 4, the total number of arguments to the expression is accumulated in num_args member of each expression class as the expression tree is built up. Leaves in the tree (objects of type ELRFad) are treated as single argument identity functions. Each binary operation such as operator * from Listing 4 treats its full set of expression arguments as the union of its two arguments, thus the operation $a \times a$ would be treated as having two arguments. The computePartials () method for each expression class computes the partial derivatives of the result of that object with respect to the expression arguments. These are stored in the partials statically allocated array, with the partials arising from the first argument in a binary operation stored in the first num_args1 locations and the partials arising from the second argument in the remaining num_args2 locations. Then the arguments of the expression tree are returned by the getArg() method allowing extraction of the derivative components of the arguments.

Listing 4 Additional expression template interface incorporating expression-level reverse mode.

```
class ETFad : public Expr<ETFadTag> {
public:
```

```
  static const int num_args = 1; // Number of expression args

  // Return partials w.r.t. arguments
  void computePartials(double bar, double partials[]) const {
    partials[0] = bar; }

  // Return argument Arg of expression
  template <typename Arg>
  const ETFad& getArg() const { return *this; }
};

template <typename T1, typename T2>
class Expr< MultTag< Expr<T1>, Expr<T2> > > {
public:
  // Number of arguments to expression
  static const int num_args1 = ExprT1::num_args;
  static const int num_args2 = ExprT2::num_args;
  static const int num_args = num_args1 + num_args2;

  // Compute partial derivatives w.r.t. arguments
  void computePartials(double bar, double partials[]) const {
    a_.computePartials(bar*b_.val(), partials);
    b_.computePartials(bar*a_.val(), partials+num_args1);
  }

  // Return argument Arg for expression
  template <int Arg> const ETFad& getArg() const {
    if (Arg < num_args1) return a_.template getArg<Arg>();
    else return b_.template getArg<Arg-num_args1>();
  }
};
```

These methods are then used to combine the expression-level reverse mode with the overall forward AD propagation through the new implementation of the assignment operator shown in Listing 5. First the derivatives with respect to the expression arguments are computed using reverse-mode AD. These are then combined with the derivative components of the expression arguments using the functor LocalAccumOp and the MPL function for_each. The overloaded operator () of LocalAccumOp computes the contribution of expression argument Arg to final tangent component i using the chain rule. The MPL function for_each then iterates over all of the expression arguments by iterating through the integral range $[0,M)$ where M is the number of expression arguments. Since M is a compile-time constant and for_each uses template recursion to perform the iteration, this has the performance of an unrolled loop.

Listing 5 Expression template forward AD propagation using expression-level reverse mode.

```
// Functor for mpl::for_each to multiply partials and tangents
template <typename ExprT> struct LocalAccumOp {
  const ExprT& x;
  mutable double t;
  double partials[Expr<T>::num_args];
  int i;
  template <typename ArgT> void operator () (ArgT arg) const {
```

```
    const int Arg = ArgT::value;
    t += partials[Arg] * x.template getArg<Arg>().dx(i);
  }
};

class ETFad : public Expr<ETFadTag> {
public:
  // ELR expression template assignment operator
  template <typename T> ELRFad& operator=(const Expr<T>& x)  {
    val_ = x.val();
    dx_.resize(x.size());

    // Compute partials w.r.t. expression arguments
    LocalAccumOp< Expr<T> > op;
    x.computePartials(1.0, op.partials);

    // Multiply partials with tangents
    const int M = Expr<T>::num_args;
    op.x = x;
    for(op.i=0; op.i<x.size(); ++op.i) {
      op.t = 0.0;
      mpl::for_each< mpl::range_c< int, 0, M > > f(op);
      dx_[i] = op.t;
    }
    return *this;
  }
};
```

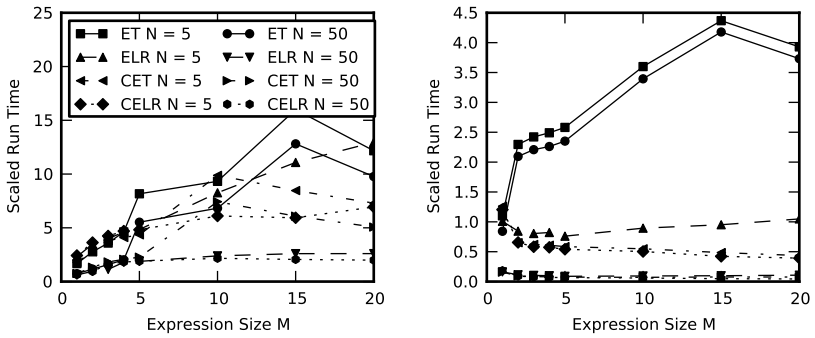
Note that as in the simple expression template implementation above, the value of each intermediate operation in the expression tree may be computed multiple times. However the values are only computed in the computePartials() and val() methods, which are each only called once per expression, and thus the amount of recomputation only depends on the expression size, not the number of independent variables. Clearly the caching approach discussed above can also be incorporated with the expression-level reverse mode approach, which will not be explicitly shown here. Again, the additional complexity necessary for constants and passive variables is not discussed here.

To test the performance of the various expression template approaches, we apply them to the simple test expressions

$$y = \prod_{i=1}^{M} x_i \quad (1) \quad \text{and} \quad y = \overbrace{\sin(\sin(\ldots\sin}^{M \text{ times}}(x)\ldots)) \quad (2)$$

for $M = 1,2,3,4,5,10,15,20$. Test function (1) tests wide but shallow expressions, whereas function (2) tests deep but narrow expressions, and together they are the extremes for expressions seen in any given computation. In Fig. 1 we show the scaled run time of propagating $N = 5$ and $N = 50$ derivative components through these expressions for each value of M using the standard expression template, expression-level reverse mode, caching, and caching expression-level reverse mode approaches

implemented in Sacado. The scaled run time is the average wall clock time divided by the product of the average undifferentiated expression evaluation time and the number of derivative components N. These tests were conducted using recent Intel and GNU compilers, run on a single core of an Intel quadcore processor. The GNU and Intel results were qualitatively similar with the GNU results shown here. With this definition of the scaled run time, one would hope to see roughly constant cost for all expression sizes. For the standard approach, this is clearly not the case with the cost increasing significantly with the size of the expression. All three of caching, expression-level reverse mode, and caching expression-level reverse mode are significant improvements, particularly for large expression sizes M or large numbers of derivative components N. Except for very small expression sizes, caching plus expression-level reverse mode appears to be the most efficient overall, however for large N the differences are not significant. We note however that only recently have compilers incorporated optimizations that yield efficient expression templates when using these more advanced approaches.



(a) Multiply function (1). (b) Nested function (2).

Fig. 1 Scaled derivative propagation time for expressions of various sizes. Here ET refers to standard expression templates, ELR to expression-level reverse mode, and CET/CELR to caching versions of these appraoches.

4 Application to Differentiation of a Fluid Dynamics Simulation

To demonstrate the impact of these approaches to problems of practical interest, we apply them to a 3D transport/reaction problem. We compute a steady-state solution to the decomposition of dilute species in a duct flow. The problem is modeled by a system of coupled differential algebraic equations that enforce the conservation of momentum, energy, and mass under non-equilibrium chemical reaction. The com-

plete set of equations, the discretization technique and the solution algorithms are described in detail in [17]. A short summary follows.

The system is discretized using a stabilized Galerkin finite element approach on an unstructured hexahedral mesh with a linear Lagrange basis. We solve three momentum equations, a total continuity equation, an energy equation and five species conservation equations resulting in 10 unknowns per basis point. Due to the strongly coupled nonlinear nature of the problem, a fully coupled, implicit, globalized inexact Newton-based solve [10] is applied. This requires the evaluation of the Jacobian sensitivity matrix for the nonlinear system. An element-based automatic differentiation approach [4, 15] is applied via template-based generic programming [13, 14] and Sacado, resulting in 80 derivative components in each element computation. The five species decomposition mechanism uses the Arrhenius equation for the temperature dependent kinetic rate, thus introducing transcendental functions via the the source terms for the species conservation equations.

Table 1 shows the evaluation times for the global Jacobian required for each Newton step, scaled by the product of the residual evaluation time and the number of derivative components per element. Both caching and expression-level reverse mode are significant improvements. Due to the large number of derivative components however, we see little difference between the three methods.

Table 1 Scaled Jacobian evaluation time for reaction/transport problem.

Implementation	Scaled Jacobian Evaluation Time
Standard expression template	0.140
Expression-level reverse	0.091
Caching expression template	0.097
Caching expression-level reverse	0.090

5 Concluding Remarks

In this paper we described challenges for using expression template techniques in operator overloading-based implementations of forward mode AD in the C++ programming language, and two approaches for overcoming them: caching and expression-level reverse mode. While expression-level reverse mode is not a new idea, we believe our use of it in operator overloading-based approaches, and its implementation using template meta-programming is unique. Together, these techniques significantly improve the performance of expression template approaches on a wide range of expressions, demonstrated through small test problems and application to a reacting flow fluid dynamics simulation. In the future we are interested in applying the approach in [12] for making the preaccumulation of the expression gradient even more efficient, and with general meta-programming techniques, this

should be feasible. For the first-order forward mode with a sufficiently large number of derivative components, this is unlikely to have a dramatic effect on performance. However for higher-order modes, such as second derivatives or Jacobian derivatives of Taylor coefficients, we would expect those techniques to become more relevant as the cost of the expression preaccumulation is much more significant.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley (2004)
2. Aubert, P., Di Césaré, N.: Expression templates and forward mode automatic differentiation. In: Corliss et al. [8], chap. 37, pp. 311–315
3. Aubert, P., Di Césaré, N., Pironneau, O.: Automatic differentiation in C++ using expression templates and application to a flow control problem. Computing and Visualization in Science **3**, 197–208 (2001)
4. Bartlett, R.A., Gay, D.M., Phipps, E.T.: Automatic differentiation of C++ codes for large-scale scientific computing. In: V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot, J. Dongarra (eds.) Computational Science – ICCS 2006, *Lecture Notes in Computer Science*, vol. 3994, pp. 525–532. Springer, Heidelberg (2006). DOI 10.1007/11758549_73
5. Bischof, C.H., Bücker, H.M., Hovland, P.D., Naumann, U., Utke, J. (eds.): Advances in Automatic Differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 64. Springer, Berlin (2008). DOI 10.1007/978-3-540-68942-3
6. Bischof, C.H., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Computational Science & Engineering **3**(3), 18–32 (1996)
7. Bischof, C.H., Haghighat, M.R.: Hierarchical approaches to automatic differentiation. In: M. Berz, C. Bischof, G. Corliss, A. Griewank (eds.) Computational Differentiation: Techniques, Applications, and Tools, pp. 83–94. SIAM, Philadelphia, PA (1996)
8. Corliss, G., Faure, C., Griewank, A., Hascoët, L., Naumann, U. (eds.): Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science. Springer, New York, NY (2002)
9. Dawes, B., Abrahams, D.: Boost C++ Libraries. `http://www.boost.org` (2011)
10. Eisenstat, S.C., Walker, H.F.: Globally convergent inexact Newton methods. SIAM J. Optim. **4**, 393–422 (1994)
11. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
12. Naumann, U., Hu, Y.: Optimal vertex elimination in single-expression-use graphs. ACM Transactions on Mathematical Software **35**(1), 1–20 (2008). DOI 10.1145/1377603.1377605
13. Pawlowski, R.P., Phipps, E.T., Salinger, A.G.: Automating embedded analysis capabilities using template-based generic programming. Scientific Programming (2011). Submitted.
14. Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Owen, S.J., Siefert, C., Staten, M.L.: Applying template-based generic programming to the simulation and analysis of partial differential equations. Scientific Programming (2011). Submitted.
15. Phipps, E.T., Bartlett, R.A., Gay, D.M., Hoekstra, R.J.: Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In: Bischof et al. [5], pp. 351–362. DOI 10.1007/978-3-540-68942-3_31
16. Phipps, E.T., Gay, D.M.: Sacado Automatic Differentiation Package. `http://trilinos.sandia.gov/packages/sacado/` (2011)
17. Shadid, J.N., Salinger, A.G., Pawlowski, R.P., Lin, P.T., Hennigan, G.L., Tuminaro, R.S., Lehoucq, R.B.: Large-scale stabilized FE computational analysis of nonlinear steady-state transport/reaction systems. Computer methods in applied mechanics and engineering **195**, 1846–1871 (2006)