filib++ , Expression Templates and the Coming Interval Standard*

M. Nehmeier and J. Wolff v. Gudenberg
Informatik 2, University of Würzburg, Am Hubland,
D-97074 Würzburg, Germany
{nehmeier,wolff}@informatik.uni-wuerzburg.de

Abstract

In this paper we investigate how a C++ class library can be improved by the concept of expression templates. Our first result is a saving of rounding mode switches which considerably increases the performance.

Our second result deals with handling the discontinuity flag that will probably be decided to be raised whenever a function is called outside its domain (loose evaluation). We discuss several alternatives and propose an expression related flag that can be used in a thread safe manner.

Both results are reviewed with respect to the coming IEEE standard for interval arithmetic.

Keywords: interval arithmetic libraries
AMS subject classifications: 65G30, 65G20, 65G40

1 Introduction

Current C++ interval libraries like Boost [1] or filib++ [6] suffer from low speed of switching the rounding mode. In filib++ rounding strategies can be instantiated by template parameters. Boost uses policy classes for different rounding strategies. In this paper we suggest an optimized rounding strategy based on expression templates.

2 Interval arithmetic using expression templates

Expression templates is a means for user-defined specification of the semantics of expressions. The expression tree is explicitly visible and can be transformed at compile time. The result is an optimized machine code. Optimization can include loop fusion [11], adaptation to grids in finite element methods [3] or increasing accuracy in dot product expressions [7]. We apply expression templates to minimize rounding mode switches.

*Submitted: January 20, 2009; Revised: January 19, 2010; Accepted: February 1, 2010.

2.1 Saving rounding mode switches

Operator overloading is used by the libraries Boost or filib++ to implement interval arithmetic. For a single operation usually three rounding mode switches are necessary, if the initial rounding mode is different from a directed rounding. See the following pseudo code in Listing 1 for the addition of three intervals $a + b + c$.

```
1  store_rounding();
2  set_rounding_downward();
3  t1 = a1 + b1;
4  set_rounding_upward();
5  t2 = a2 + b2;
6  reset_rounding();
7  store_rounding();
8  set_rounding_downward();
9  r1 = t1 + c1;
10 set_rounding_upward();
11 r2 = t2 + c2;
12 reset_rounding();
```

Listing 1: Addition of three intervals $a + b + c$

Three of the six rounding mode switches can be saved by a simple rearrangement.

```
1  store_rounding();
2  set_rounding_downward();
3  t1 = a1 + b1;
4  set_rounding_upward();
5  t2 = a2 + b2;
6  r2 = t2 + c2;
7  set_rounding_downward();
8  r1 = t1 + c1;
9  reset_rounding();
```

Listing 2: Rearranged addition of three intervals $a + b + c$

A further optimization by moving the computation of r1 after line 3 is not possible for all operations. E.g., for multiplication both bounds of the first product have to be known before the second product can be computed.

Another optimization is to apply the formula $\triangle x = -\triangledown(-x)$.

```
1  store_rounding();
2  set_rounding_downward();
3  t1 = a1 + b1;
4  t2 = -(-a2 - b2);
5  r1 = t1 + c1;
6  r2 = -(-t2 - c2);
7  reset_rounding();
```

Listing 3: Addition of three intervals $a + b + c$ with one rounding mode

Many compilers optimize the computation of the upper bounds in line 4 and 6 to a simple addition now being performed with the wrong rounding mode. This can be avoided by storing the upper bound negative [5].

2.2 Evaluation of expression trees

During compilation time the following expression tree is instantiated for the addition of three intervals. The evaluation is delayed until assignment of the complete expression.

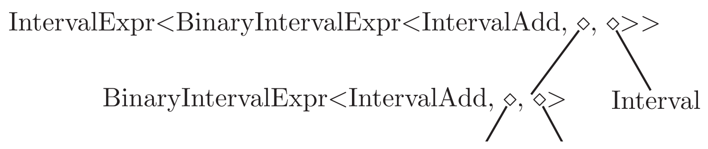
IntervalExpr<BinaryIntervalExpr<IntervalAdd, ◇, ◇>>
BinaryIntervalExpr<IntervalAdd, ◇, ◇>
Interval

Figure 1: Expression tree for the addition of 3 intervals

The class diagram in Figure 2 displays the behavior of the expression template class for a binary operation. The two expression operands as well as the operator are used to instantiate the template, see Figure 1. This class represents the expression with the evaluation function which is called with the actual rounding control. The rounding control visits the whole tree[1], hence, it can be accessed by each part of the expression. Depending on the current state a decision whether to switch the rounding mode can be taken. Note that the classes `IntervalExpr` and `BinaryIntervalExpr` are transparent to the user.

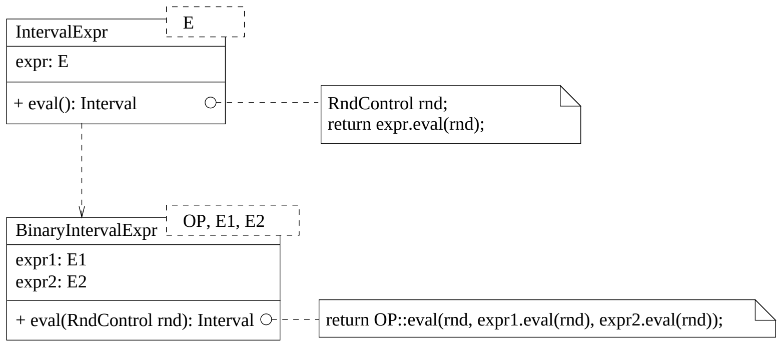

Figure 2: Class diagram for `IntervalExpr`

The expressions are not restricted to elementary operations but can contain interval function calls. These are evaluated before the rest of the tree. Mixed mode arithmetic with floating point operations may be supported.

The reduction of rounding mode switches for n operations is displayed in table 2.2.

[1]That is called a *visitor pattern* [2] in software engineering.

rounding strategy	operator overloading	expression templates
switched	$3n$	$n+2$
onesided	$2n$	2

Table 1: Number of rounding mode switches

2.3 Performance tests

For performance tests the concept of expression templates was implemented with the two rounding strategies “switched” and “onesided”, see Listing 2 and 3, respectively. The implementation uses plain C++ code in contrast to the current filib++ implementation that uses assembler subroutines. Therefore the existing filib++ implementation is the fastest for short expressions. For longer expressions with three or more operations our test implementation with expression templates results in a measurable increase of performance. If we apply the same “tricks” as filib++, we can further reduce the runtime.

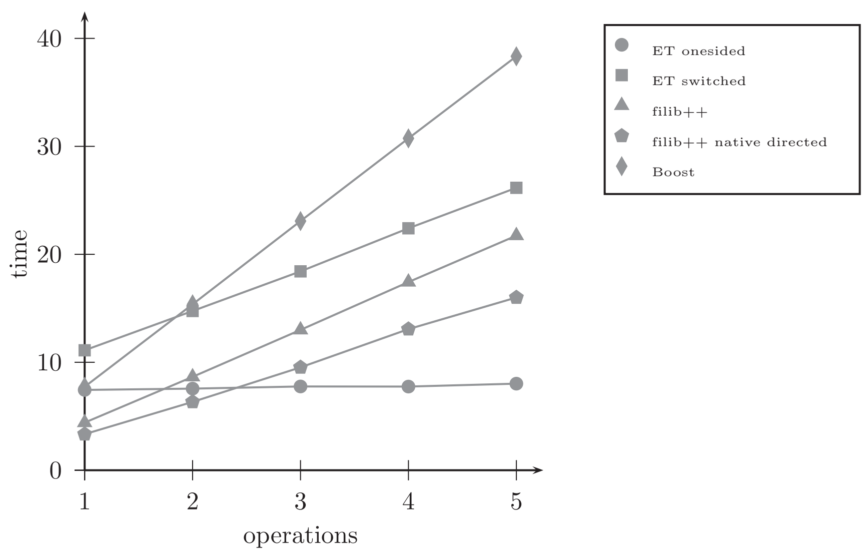

Figure 3: Performance

We recommend not to keep intermediate results in temporary variables, since the gain of performance is higher for longer expressions.

3 Flag Handling

For loose evaluation of functions or division by an interval containing zero flags will have to be set according to the standard. These flags indicate that a function has been called with an argument outside the domain of definition. In either case continuity is not valid anymore. Hence, assertions needing continuity like Brouwer's fixed point theorem can not be applied.

The flags have to be checked by the user. There are several ways to implement these flags.

3.1 Global flags

Intervals in filib++ can be called with different modes. The standard or interval mode does not allow loose evaluation and terminates the program, if a call outside the domain occurs. No flags are provided. In the extended mode filib++ returns the containment set, i.e. an element of $\mathbb{IR}^*$. Again no flags occur. In the third mode, extended with flag, a global flag is maintained. It has to be reset by the user and is set, if an operation or function is called outside its domain. A global flag, however, has several disadvantages. It is not possible to mark the expression that was responsible for setting the flag and, more crucial, thread safety can not be guaranteed in modern multi-scalar, multi-core computer architectures.

3.2 User managed flags

For the proposed interval arithmetic part of the C++ standard library[9] an explicitly managed flag – completely under user control – had do be provided, since it was required that the compiler does not need to know anything about the flag. In other words the flag handling has to be transparent for the compiler. This concept overcomes the first disadvantage of global flag handling.

The user has to call extended functions and "operators" that receive and deliver the flag as an explicit boolean variable. The code completely looses readability:

```
Interval x(1), y(-1,1);

bool f = false;
Interval r = divide(x , sqrt(y,f) , f);
if (!f)
        ...
```

Listing 4: $r = x/\sqrt{y}$ user defined flag

Furthermore, since the user can reset the flags, it is difficult to retain thread safe code.

3.3 Expression related flags

Our proposal is to use local flags for each interval expression. That can be applied in the same manner as the C++ library flag without having to be managed by the user. This is achieved by the introduction of the new data type `ExpressionResult` that keeps an expression local flag.

`ExpressionResult` is a data type that contains an interval and several flags, therefore an alternative name would be `FlaggedInterval`. It can be passed as an operand to expressions, then the flag is also included. If, however, the `ExpressionResult` is converted to an interval the flag is disregarded.

Since one or even more flags per interval would not only boost the representation of an interval to 2 doubles plus some bits and, hence, do not fit in any reasonable ports and buses but also slow down the operations, we suggest to provide the new data type for exception handling during evaluation and use plain intervals as storage format.

This has the following advantages. If the information that is indicated by the flag is important for the application, the user just assigns the return value of an expression to a variable of the type `ExpressionResult`. That is possible, since `ExpressionResult` is assignment compatible with `Interval`. For a user who doesn 't care about flag and therefore doesn't use the type `ExpressionResult` the evaluation corresponds to the extended mode in filib++.

```
Interval x(1), y(-1,1);

ExpressionResult er = x/sqrt(y);
if (!er.getExtendedErrorFlag())
        Interval r = er;
        ...
```

Listing 5: $r = x/\sqrt{y}$ expression related flag

Implementation of this data type can be done by operator overloading or by expression templates. In the latter case the `eval()` method can use traits for the evaluation of different types.

```
template<typename OP, typename E1, typename E2>
class BinaryIntervalExpr {
  private:
    typedef EvalTraits<E1>  et1;
    typedef EvalTraits<E2>  et2;

    E1 _expr1;
    E2 _expr2;

  public:
    BinaryIntervalExpr(E1 expr1, E2 expr2)
        : _expr1(expr1), _expr2(expr2) {}

    Interval eval(RndControl& rnd,
            FlagControl& flag) const {
        return OP::eval(rnd,
            flag,
            et1::eval(rnd, flag, _expr1),
            et2::eval(rnd, flag, _expr2));
    }
};
```

Listing 6: Traits Implementation of the class `BinaryIntervalExpr`

Walking through the tree for evaluation the flag is passed as a visitor object. The different `eval()` methods can set the flag if necessary.

```
template<typename E>
class IntervalExpr {
  private:
    E _expr;

  public:
          ...
    ExprResult eval() const {
        RndControl rnd;
        FlagControl flag;
        return ExprResult(_expr.eval(rnd, flag), flag);
    }
};
```

Listing 7: Evaluation of an expression tree

Using the expression template implementation an evaluation of an expression is delayed until all information is available as a tree. The flag of the whole expression is computed by or-ing the flags of the subexpressions. Hence, it does not matter in which order the computations of the subexpressions arrive.

For the implementation with operator overloading the flags only occur as result parameters and again the order of evaluation does not change the final value of the flag. So we have the following proposition.

Proposition: The handling of expression local flags is thread safe.

4 Coming interval standard

Currently interval arithmetic is being standardized by the IEEE working group P1788. The definition of interval arithmetic will be described in levels of abstraction [10]. Level 1 is the application level whereas level 2 defines the interval operations. Level 3 is responsible for the representation of interval data, and level 4 finally specifies the bit strings. Level 2 may be regarded as the interface of the abstract data type interval specifying its constructors and operations.

It is still under discussion whether level 2 should be implemented in hardware or software. In our opinion the whole standard is to be formulated independent from such issues. We even think that at least level 4, the bit layout level should not be specified completely.

Each implementation, however, in hardware or software has to provide all operations of level 2. Combined hardware and software solutions are possible. Figure 4 displays different levels of abstraction. Starting from the level 2 interface of the P1788 standard, there is a choice of an implementation in a particular programming language using the instruction set of an existing processor or a new interval processing unit. This level can be refined by choosing various implementation options.

We hope that our results will have some influence on the discussion of that standard.[2]

[2]In the meantime the standard provides exception handling with decorated intervals which may be considered as extension of our flagged interval concept.

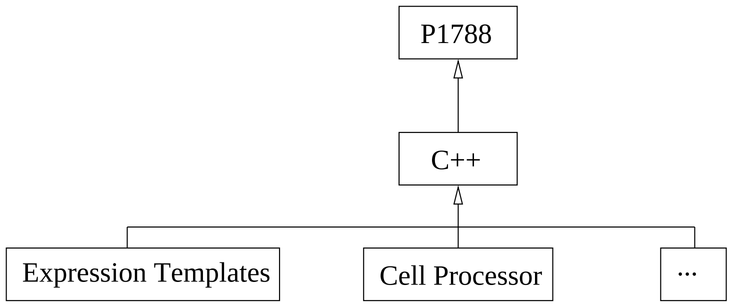

Figure 4: Levels of abstraction

5 Summary

We have used expression templates to compute interval expressions. On the one hand we could save rounding mode switches and thus accelerate the performance. On the other hand we discussed thread safe flag handling for loose evaluation. It can be established for expression related flags. This is achieved by the introduction of the new data type `ExpressionResult` that contains an interval and several flags. The flag handling can be applied to expressions computed by several statements, if we use the data type `ExpressionResult` for intermediate values.

References

[1] Boost Interval Arithmetic Library, January 2009.
`http://www.boost.org/doc/libs/1_37_0/libs/numeric/interval/doc/interval.htm`

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] J. Härdtlein, *Moderne Expression Templates Programmierung*, PhD thesis, Universität Erlangen-Nürnberg, 2007, in German.

[4] *IEEE Interval Standard Working Group – P1788*, January 2009.
`http://grouper.ieee.org/groups/1788/`

[5] B. Lambov, "Interval arithmetic using SSE-2", *Lecture Notes in Computer Science*, vol. 5045, pp. 102–113, 2008.

[6] M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster, and W. Krämer. "Filib++, a fast interval library supporting containment computations", *ACM Trans. Math. Softw.*, vol. 32, no. 2, pp. 299–324, 2006.

[7] M. Lerch and J. Wolff von Gudenberg, "Expression templates for dot product expressions", *Reliable Computing*, vol. 5, no. 1, pp. 69–80, 1999.

[8] S. B. Lippman, ed., *C++ Gems*, SIGS Publications, Inc., New York, NY, USA, 1996.

[9] S. Pion, H. Brönnimann, and G. Melquiond, "A proposal to add interval arithmetic to the C++ standard library" In: P. Hertling, C. M. Hoffmann, W. Luther, and N. Revol, editors, *Reliable Implementation of Real Number Algorithms: Theory and Practice*, Dagstuhl Seminar Proceedings, no. 06021, Schloss Dagstuhl, Germany, 2006. `http://drops.dagstuhl.de/opus/volltexte/2006/718`

[10] J. Pryce and D. Lester, *A proposed structure for the process of constructing the P1788 standard*, December 2008, in [4].

[11] T. L. Veldhuizen, "Expression templates", *C++ Report*, vol. 7, no. 5, pp. 26–31, June 1995; Reprinted in [8].