C++ Expression Templates Performance Issues in Scientific Computing

Federico Bassetti, Kei Davis, and Dan Quinlan

CRPC-TR97705-S
October 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997

C++ Expression Templates Performance Issues in Scientific Computing

Federico Bassetti* Kei Davis Dan Quinlan†

Abstract

Ever-increasing size and complexity of software applications and libraries in scientific computing is making implementation in the programming languages traditional for this field—FORTRAN 77 and C—impractical. The major impediment to the progression to a higher-level language such as C++ is attaining FORTRAN 77 or C performance, which is considered absolutely necessary by many practitioners. The use of template metaprogramming in C++, in the form of so-called expression templates to generate custom C++ code, holds great promise for getting C performance from C++ in the context of operations on array-like objects. Several sophisticated expression template implementations of array-class libraries exist, and in certain circumstances their promise of performance is realized. Unfortunately this is not uniformly the case; this paper explores the major reasons that this is so.

1 Introduction

Scientific computing, though traditionally lagging up-to-date software development and engineering practices, is nonetheless undergoing a rapid evolution. In particular large and complex applications and software infrastructures are increasingly enjoying the benefits of object-oriented design and implementation in C++. In at least one respect the object-oriented approach to scientific computing yields benefits not usually realized in more mainstream computing: management of data distribution, parallel computation (often SPMD), and communications can be incorporated into class libraries and so hidden from the library user.

This transition is not without problems or opponents. While simple inertia can in part be blamed for the slow pace of change (and the fact that perhaps the majority of scientific programmers are physical scientists or mathematicians with little or no formal computing science background), in scientific computing the very highest priority is most often performance, and the performance penalties generally associated with higher-level languages or language constructs is usually deemed unacceptable. Thus while the software

*Bassetti: Computer Science Dept., New Mexico State University, Las Cruces, NM 88003.

†Bassetti, Davis, and Quinlan: Scientific Computing Group CIC-19, Computing, Information, and Communications Division, Los Alamos NM, USA, 87545, {fede,kei,dquinlan}@lanl.gov

industry at large enthusiastically adopted C++ as an improvement over C and is increasing using Java, scientific computing has been moving from FORTRAN 77 to C; significant progress to C++ will depend crucially on whether C++ can deliver the performance of carefully coded C or FORTRAN 77.

The sheer size and complexity of the scientific applications being developed here at LANL and elsewhere has practically mandated development in a higher-level language such as C++. We are therefore heavily invested in its continued use and greater acceptance and so in realizing 'optimal'—C or FORTRAN 77—performance. A current 'hot' area is the use of the C++ templating mechanism [8] to implement so-called *expression templates (ETs)* [10] to get automatic code in-lining and fusion of the loops implicit in expressions denoting (multiple) array operations that would more traditionally be implemented using overloaded binary operators, in turn implemented by multiple function calls. This approach has much promise and quite sophisticated expression template implementations have been developed (e.g. [9]). Unfortunately this technique is not without its deficiencies, some of which may be regarded as artifacts of the current state of the art in C++ compiler technology (such as incomplete support and astronomical compile-time and space requirements); others appear to be inherent in the technique, consequences of lack of information needed by the compiler to perform standard optimizations. It is in the possibly intrinsic limitations that we are interested.

The study we describe was motivated by the performance of various state-of-the-art expression template implementations of array and array-like C++ class libraries falling short of expectations. At this stage of investigation we do not claim that the causes we explicate are insurmountable. Rather, the points are as follows: these are the first analyses of this depth, and, analysis at this depth is necessary to pinpoint the sources of the problems, suggest possible solutions, and ultimately precisely characterize the trade-off between theoretical limitations on performance and practical limitations on implementability and usability of the ET technique.

2 Problem Domain and Execution Models

Objects with array-like semantics are fundamental to scientific computing, and numerous sophisticated array-class libraries have been implemented and are in heavy use, e.g. A++/P++ [5] and POOMA [6], often underneath higher-level C++ class libraries such as OVERTURE [1], which add support for complex geometry, adaptive mesh refinement, moving grids, and other features to meet more sophisticated applications requirements. As such the performance of the latter libraries and the applications that use them is directly related to that of the underlying array-class libraries.

The standard technique for implementing array class libraries in C++ is to overload binary operators such as `+` to denote for each type of array element, dimensionality and size of array, etc., the corresponding array operation, e.g. elementwise addition, implemented as a class member function call. Thus an array expression such as `A+B+C` entails two function calls and in turn two loops—one to add `A` and `B`, and one to add the result to `C`. Moreover, a temporary array must be created for each intermediate result, and

subsequently destroyed. Using expression templates these three sources of inefficiency may be avoided—the expression template mechanism essentially specifies the C++ code to be generated; typically function calls are inlined and a single loop generated. Depending on the semantics and dependence of left- and right-hand side only one or zero temporaries need be created. The expectation is for the performance of hand-written C code, and in certain circumstances this is realized. Our interest is in the circumstances and the underlying reasons when it falls short.

In brief, the greatest contributors to performance loss are the various consequences of demand for CPU registers. The second greatest cause—poor blocking—is not addressed here.

3 Methodology

For various forms of expressions typical of scientific applications using array class libraries we compare the performance of three implementations: overloaded binary operators, the natural implementation in C++ code (in C style, not using overloaded binary operators) and an idealized form of the code generated using expression templates (which we call *emulated expression template (EET)* code). The EET code is idealized in that it is somewhat simplified to eliminate extraneous effects and implementation-dependent irrelevancies while maintaining the structure that gives the ET code its particular performance characteristics. In all cases the EET code gives an upper (best) bound on performance of actual ET code.

Tests were performed on an SGI Origin 2000 system [3]. The Origin 2000 uses the MIPS R10000 microprocessor [11] which has built-in hardware performance counters for collecting run-time statistics for arbitrary sections of code. Using these counters we measured the number of cycles executed, instructions executed, floating point operations, primary and secondary cache misses, load and store instructions executed, and several other parameters. While most of these results are not reported the statistics were monitored to ensure that our explanations of execution times were correct. Some of the metrics reported are computed from combinations of counters [7, 4, 2]. All tests were for serial codes on a single processor within that processor's physical segment of the distributed shared memory, so details of the multi-processor environment are irrelevant. In is important to stress that problem sizes were chosen to be L1-cache resident to avoid cacheing issues.

The benchmarking results were generated using the KAI C++ compiler and were verified using the SGI C++ compiler. The KAI compiler translates C++ code to intermediate C code and then invokes an independent C compiler (here the SGI C compiler). We present both C++ codes and the intermediate C codes produced from them. Examination of assembly code produced by the C compiler was useful for determining the causes of performance loss, but its presentation is not necessary for our demonstrations.

Stencil-like computations represent a significant part of scientific codes. We address these in particular, as well as other common patterns of array use. This paper explores a space of array statements and reports on their performance. The primary dimensions of this space are number of dimensions and number of operands. In many cases the

dependence on dimension is not significant while the number of operands is more uniformly important. We have chosen the number of operands from 1-61 since this is sufficient to display the relavant character of the problems we have exposed. Central to our thesis is that for stencil operations array statements with this number of operands are common within our work using OVERTURE. OVERTURE includes specialized support for complex geometry and applications using such support have numerical discritizations which include cross-derivative terms which contribute to the number of points included in a stencil. Additionally such applications include non-constant coefficents which are themselves stored in arrays and so can effectively double the number of arrays (operands) used in an array statement. Higher-order discretizations fold in even more points into stencils so that 3D stencils can have 125 operands, still not counting coefficients which are additional arrays along with scalar coefficients. A commonly used operator within OVERTURE applications, as a specific example, contains 130+ array operands, not including numerous scalar values. For completeness we also test the case of all operands are from distinct arrays but do not claim that this is representative of typical numerical computations.

4 Test 1: Number of operands in a stencil-like code

The two versions of the test code implement a simple 3-pt stencil using unidimensional arrays. The core of the computation is a loop that traverses the elements of the arrays in memory order; it is repeated 10 times to ensure the accuracy of the performance data. An instance of the code is given in Figure 1.

The main differences between the two codes are in the way the arrays are accessed. In C++ the array on the right-hand-side is reused in all the terms, and this is true regardless of the number of operands. Thus register requirements are constant. A code that makes use of expression templates will use a different array pointer for each operand (despite the fact that they have the same value), and also carries information on how to compute the proper offset. (There is work underway to remove this latter problem, but at best this would reduce register demand by one-half.) Thus register requirements are twice the number of operands. A more clear idea of the whole transformation can be gained by looking at the intermediate C code generated by a compiler—together with a different pointer per operand, a different stride is also needed. Fundamentally, the different pointers are generated because the indexing operators generate different return values for each invocation, and for more complex reasons (because of the subtleties of ET implementation) even when there is no indexing. Run-time optimizations can be used to avoid the additional stride information at the cost of extra code, but this adds a substantial penalty to (already excessive) compile times and is not done in any of the production array classes.

4.1 Measurements and Results

Register spillage refers to the circumstance in which demand for registers exceeds the number available, resulting in register values being stored to memory and subsequently

```
// C++
for (int iter = 0; iter < cntmax; iter++)
    for (int i = 2; i < size; i++)
      A[i] = B[i-1] + B[i] + B[i+1];

// Emulated Expression Templates
Array B_1=B;
Array B_2=B;
Array B_3=B;
for (int iter = 0; iter < cntmax; iter++)
    for (int i = 2; i < size; i_1++)
      A.DataPointer[A.offset(i)] = B_1.expand(i-1) + B_2.expand(i) + B_3.expand(i+1);

//KCC intermediate C code for C++ code
do {
  auto int i =2;
  for (; i < size; i+=1)
      A[i] = B[i-1] + B[i] + B[i+1];
  iter += 1;
} while ( iter < cntmax);

//KCC intermediate C code for Emulated Expression Templates code
AA = A.DataPointer;
BB_1 = B_1.DataPointer;
BB_2 = B_2.DataPointer;
BB_3 = B_3.DataPointer;
int s_0 = A.Stride;
int s_1 = B_1.Stride;
int s_2 = B_2.Stride;
int s_3 = B_3.Stride;
do {
  auto int i =2;
  for (; i < size; i+=1)
      A[i*s_0] = B_1[(-1+i)*s_1] + B_2[i*s_2] + B_3[(1+i)*s_3];
  iter += 1;
} while ( iter < cntmax);
```

Figure 1: Test code 1—1D 3pt stencil

reloaded. The impact of register spillage is determined by analyzing the number of loads and stores performed. The actual number of loads and stores is determined using the performance counters and the annotated assembly code shows what kind of registers are needed (integer or floating point) and what optimizations have been enabled or disabled.

Figure 2 shows that the execution time, measured in cycles, is significantly different between the two codes. The surfaces start diverging from a single operand. As the number of operands increases the number of loads for C stays constant while for ETs it increases linearly. The figure shows that the number of loads increases mildly until the number of operands is in the range 21–26. On the MIPS R10000 there is a maximum of 27 integer and 27 floating-point registers available for general use. The assembly code shows that it is only the integer register set that spilled. Fewer floating point registers are required because of the scheduling of the operations. Demand for registers from each set is shown in Table 1.

For both number of cycles and loads the figure shows three slopes for EET code. For up to 4 operands EET performance is very close to C performance. Between 5 and 13

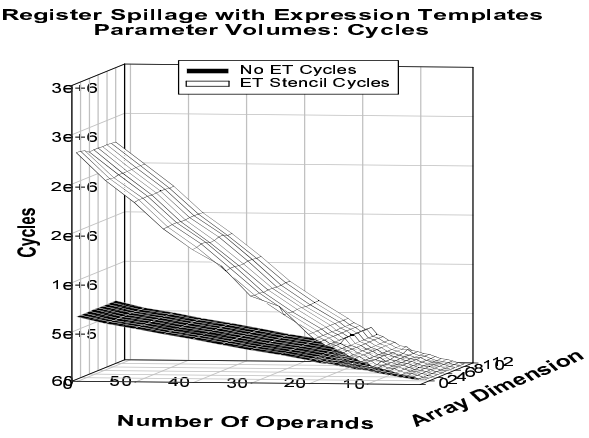


Figure 2: Measured cycles using a stencil computation

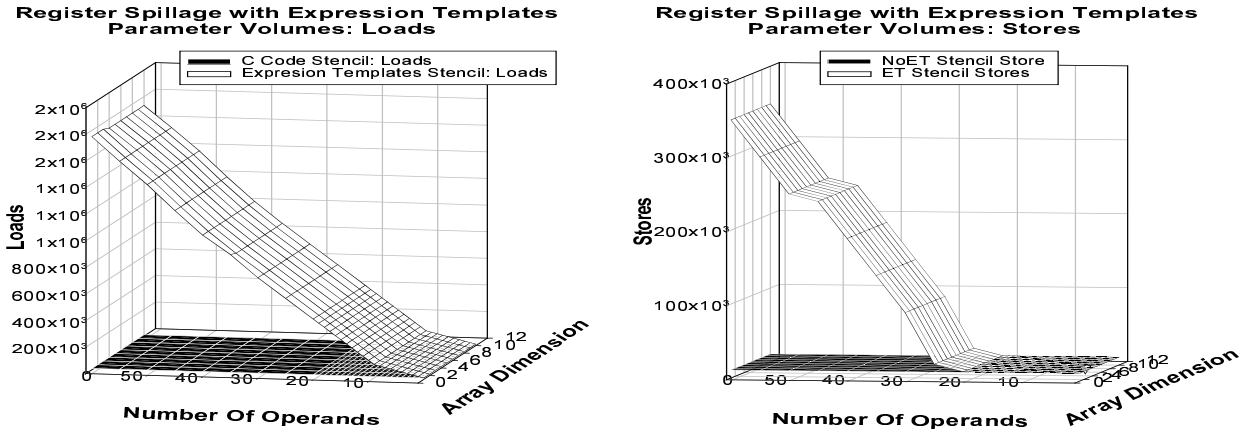
operands the EET performance degrades more rapidly; in this range loop unrolling is inhibited to prevent register spillage. From 14 to 26 (and onward) the slope is steeper; here pipelining is also inhibited and there is register spillage.

An increase in memory references appears to be the primary overhead of ETs as the number of operands grows. The figures show that the number of loads increases rapidly once registers begin to spill. Surprisingly, the number of stores is also increased. In particular the result of the subscript computation needs to be stored on the stack and not kept in a register. It is clear that ETs stress the management of integer registers. Floating point registers are well managed and demand is actually reduced by the spilling of integer registers.

Table 2 presents data collected from the annotated assembly for a pure C code. The table shows the same information as Table 1. Unrolling and pipelining are the two main optimizations that make a significant impact on the performance of the codes. The data show that the software pipeline is always possible since integer registers are never spilled. Unrolling, for the same reason, is possible for a higher number of operands. The demand for floating point registers is instead slightly higher than the expression templates case. In the C code, the number of integer and floating point registers needed is of the same order, this is because more efficient use can be made of the floating point register set.

5 Test 2: Dimensionality in a stencil-like code

In this test the impact of increasing the dimensionality of arrays in a stencil-like code is examined. The test code is similar to that in the previous section except that the dimension of each array is varied rather than the number of operands. The goal is to quantify the overhead introduced by the descriptors associated with each array. Figures 4 and 5 give the code for the 2D test cases and the corresponding intermediate C codes for



(a) Loads.

(b) Store.

Figure 3: Register Spillage effects in a stencil code

C++ and EET, respectively.

The intermediate representations of the two codes show that the offsets needed in the innermost loop are computed in the next outer loop. In particular, this computation is a function of the number of elements in each dimension.

5.1 Measurements and Results

Figure 6 shows the impact on performance as a function of dimensionality. For C there is no effect. For EET increasing dimensionality degrades performance. As before the problem is related to the extra memory references generated as consequence of a high demand for registers. It can be seen from the intermediate representation that increasing the dimensionality increases the computation needed to perform the subscript computation—more variables are employed as the dimension grows. The effects caused by multidimensional arrays are in addition effects caused by the increase in number of operands. Figure 6 shows the effect on the number of cycles as the dimensionality increases, for 6D arrays extra overhead is about 60% over 1D. In comparison the consequence of increasing dimensionality is considerably less than increasing the number of operands; in perspective, for stencil-like codes the number of operands typically increases more rapidly than the dimension.

6 Test 3: Number of operands in a non-stencil code

The previous tests have shown the impact on performance of duplication of pointers and stride variables. Here the duplication of pointers is not an issue since all the operands on the right-hand-side are distinct. The test codes are shown in Figures 7 and 8. The

number of operands	fixed point registers used	floating point registers used	software pipelining	iterations unrolled
1	15	1	ON	4
2	12	2	ON	2
3	16	4	ON	2
4	12	3	ON	0
5	14	3	ON	0
6	16	4	ON	0
7	18	3	ON	0
8	20	4	ON	0
9	22	4	ON	0
10	24	4	ON	0
11	24	4	ON	0
12	27	5	ON	0
13	27	3	ON	0
14			OFF	0

Table 1: Integer and floating point registers demand and their impact on pipelining and unrolling using Expression Templates.

intermediate C codes are very similar. The only difference is that the ET code requires an offset variable for each operand. As noted previously, such offsets can in some cases be eliminated.

6.1 Measurements and Results

Figure 9 shows the number of cycles for each code. The performance of the two codes are close though the EET code is slower. A notable point is that the C++ code shows an increase in cycles per operand added. The slope is shallow between 1 and 4 operands. The slope is greater between 5 and 26 operands because loop unrolling is inhibited, though software pipelining is still active. From 26 to 36 operands the compiler manages to shedule the instruction in such a way that it is still possible to pipeline but with slightly higher demand for registers than are actually available. However, this 'forced' pipelining doesn't appear to have a positive impact on performance. From 41 operands and up, the slope becomes again linear in the number of operands. Table 3 shows for C++ the demand on integer and floating point registers and the corresponding impact on software pipelining and loop unrolling. For ET code the numbers are exactly the same as shown in Table 1. Keeping in mind how and where the two codes differ, the impact of having a more complicated subscript computation can be quantified for ETs. The behavior shown for cycles is reflected by the number of loads performed. It is that the ET code has a higher demand for registers since the software pipeline is turned off at 13 operands, while for the C++ code that drop in performance comes at 26 operands. The more complicated subscript computation, once registers are spilled, is the cause of an increase in store instructions as show in Figure 10. The difference in the three metrics (cycles, loads, stores) quantifies the effects of duplicating offset variables in the case of ETs. Note that for fewer than 13 operands the performance of the two codes are identical.

The test codes presented in this section, also help to determine parameterized ar-

number of operands	fixed point registers used	floating point registers used	software pipelining	iterations unrolled
1	5	12	ON	4
2	5	15	ON	4
3	6	18	ON	4
4	6	10	ON	2
5	6	10	ON	2
6	6	9	ON	0
7	8	9	ON	0
8	7	9	ON	0
9	8	9	ON	0
10	7	8	ON	0
11	7	8	ON	0
12	7	9	ON	0
13	7	8	ON	0
14	7	9	ON	0
15	7	8	ON	0
16	7	9	ON	0
17	7	8	ON	0
18	7	9	ON	0
19	7	8	ON	0
20	7	9	ON	0
21	7	8	ON	0
26	7	9	ON	0
31	7	8	ON	0
36	7	9	ON	0

Table 2: Integer and floating point registers demand and their impact on pipelining and unrolling using C++ code.

eas/volumes of performance spaces. In fact, putting together the results obtained for test code 1, we can first determine an upper bound and a lower bound for a class of C++ codes, and an upper bound and a lower bound for the equivalent class of codes making use of expression templates.

7 Test 4: Number operands in a stencil-like code with binary operators

Here the investigation of the variation in performance as a function of the number of operands when using binary operators. The results clarify the properties of binary operators, but since the test data fit into cache they do not indicate the performance problems with binary operators on stencil-like statements where reuse of operands is high wherein the cache is flushed and performance is about 25% of optimal for optimized implementations. The code, which emulates the behavior of binary operators in a 1-D stencil, is given in Figure 7. The emulation gives an upper bound on possible performance. To prevent loop fusion, conditional statements are interposed between the loops—in 'real' code other code is so interposed to the same effect. The comparison is between this code and the two codes in Test 1.

```
// C++ code
for( int iter = 0; iter < cntmax ; iter++ )
  for( int j = 0; j < jsize; j++ )
    for( int k = 2; k < ksize - 3; k++ )
       A[(j * ksize) + k] = B[(j * ksize) + k] + B[(j * ksize) + k + 1] +
                            B[(j * ksize) + k - 1];


// KCC intermediate C code for C++ code
do  {
   auto int j = 0;
   do  {
     auto int k = 2;
     sss = j * ksize;
     for (; (k < ksize-3); k += 1)
         A[sss + k] = B[sss + k] + B[1 + (sss + k)] + B[(-1) + (sss + k)];
     j += 1;
   } while (j < jsize); }
} while (iter < cntmax); }
```

Figure 4: Test code 2—C++ 2D 3pt stencil

7.1 Measurements and Results

Here register spillage does not differentiate performance. Figure 12 shows the number of cycles for all three codes. The slopes are all constant. From the numbers of loads and stores as shown in Figure 13 it may be inferred that 'caching' of memory in registers is minimal. Worse, extra memory references are generated by the introduction of a temporary array. Looking at just stores one can see the impact of having to store values in a temporary, while in a regular C++ code only a number of stores close to the size of the array is needed.

More interesting is the comparison between binary operators and expression templates. From Figure 12 one can see that expression templates performs better than binary operators until approaching a point where the number of operands causes register spillage. In this region the performance of the two are equivalent, but from this point on the expression templates performs worse than binary operators. The justifications for this effect are shown by the number of loads. Register spillage is a more dramatic effect than just a naive way of using the registers as in the case of binary operators. Nevertheless, the difference in cycles is not as big as the one for loads the extra stores generated by binary operators reduce the negative impact of extra loads for expression templates. Again, use of binary operators would force reloading of cache for problems exceeding cache size.

8 Test 5: Number of operands in a non-stencil code using binary operators

In this section we study the performance of binary operators on a non-stencil code, such as test code 3. Also, we present a comparison of binary operators with expresion templates and C++ code. Figure 4 shows the test code that emulates binary operators in a non-

```
// Emulated Expression Templates
for( int iter = 0; iter < cntmax; iter++ )
  for( int j = 0; j < jsize; j++ )
    for( int k = 2; k < ksize - 3; k++ )
         A.DataPointer[A.offset(k,j)] = B_1.expand(k,j) +
                                        B_2.expand(k+1,j) + B_3.expand(k-1,j);

// KCC intermediate C code for Emulated Expression Templates
AA = A.DataPointer;
joffset_0 = A.Stride[0];
BB_1 = B_1.DataPointer;
joffset_1 = B_1.Stride[0];
BB_2 = B_2.DataPointer;
joffset_2 = B_2.Stride[0];
BB_3 = B_3.DataPointer;
joffset_3 = B_3.Stride[0];
do  {
 auto int j = 0;
 do  {
   auto int k = 2;
   koffset_0 = j * A.Stride[1] * A.Size[0];
   koffset_1 = j * B_1.Stride[1] * B_1.Size[0];
   koffset_2 = j * B_2.Stride[1] * B_2.Size[0];
   koffset_3 = j * B_3.Stride[1] * B_3.Size[0];
   for (; (k < ksize_1); k += 1)
     AA[i * joffset_0 + koffset_0] =
          BB_1[k * joffset_1 + koffset_1] +
          BB_2[(1 + k) * joffset_2 + koffset_2] +
          BB_3[((-1) + k) * joffset_3 + koffset_3];
   j += 1;
  } while (j < jsize);
  i += 1;
 } while (i < isize);
 iter += 1;
} while (iter < cntmax);
```

Figure 5: Test code 2—EET 2D 3pt stencil

stencil computation. This test code has the same characteristics of the code used in the previous section. The main difference is that the minimal fraction of reuse, present in a stencil-like code, is now absent, since every operand is different from each other. In our benchmarking approach we evaluate the performance of binary operators against expression templates and C++, increasing the number of operands. Again the caveat regarding binary operators and cache residency is applicable.

8.1 Measurements and Results

From Figure 12 one can see that cycles for binary operators have a slope that is linear with the number of operands. As discussed in the previous section, extra memory references are the reason for the behavior of the binary operators as shown by the surfaces for loads and stores in Figure 13. The comparison of binary operators with C++ code has two main parts. A first part in which the C++ code can use registers to avoid unnecessary memory references. A second part in which, after spilling registers C++ and binary operators have roughly the same slope. The volume between the two surfaces is due to the extra stores and

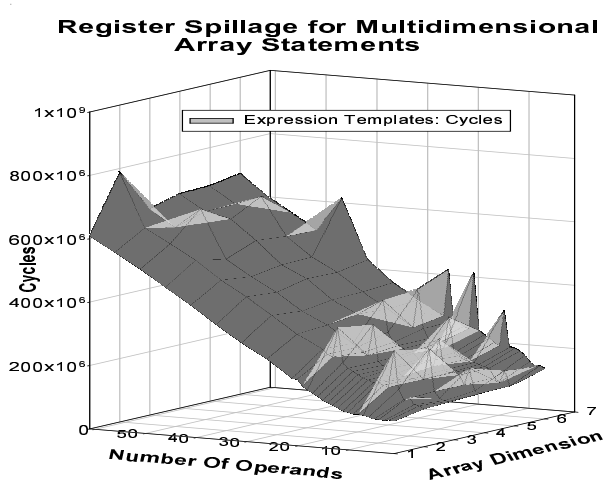


Figure 6: Measured cycles for an EET multidimensional stencil

loads as can be seen in Figure 13. The comparison with expression templates has again two main parts. The first one similarly to the C++ case is in favour of expression templates that manages to better use registers to avoid extra memory references. However, in the second part, once registers are spilled binary operators performs better than expression templates. As can be observed from Figure 13, an excess number of loads is the reason for the difference in performance. Notice that while C++ code and expression templates are spilling registers for some number of operands, binary operators never incurr this type of problem.

9 Conclusions

We have shown that artifacts of ET implementation lead to demand on registers greater than that of C code with consequent performance penalties. In doing so we have attempted to define in general terms the parameter spaces of array statements and their perfromance on the Origin 2000, a single machine but a critically important one for ASCI work at LANL. Having so clearly related the performance data to the number of registers in the machine we expect that such results can be parameterized by the number of registers of any machine with predictable results. To the best of our knowledge this is the first such in-depth study and analysis of performace issues of optimization mechanisms for object-oriented array class libraries to date, encompassing expression templates, binary operators, and C code as might be hand written or produced using alternative optimization mechanisms such as source-to-source transformation tools.

We do not claim that the defects of ET implementation are intrinsic to the technique, but they certainly are present in all state-of-the-art implementations of which we are aware. In any case the results of this work have already stimulated closer examination of the potential of ET techniques to achieve C performance, in both one of the author's

```
// C++
for( int iter=0; iter <cntmax ; iter++ )
  for( int i = 2; i < size-3; i++ ){
    A[i_1] = B_1[i] + B_2[i+1] + B_3[i-1];

//Emulated Expression Templates
Array<1> A(SizeArray);
Array<1> B_1(SizeArray);
Array<1> B_2(SizeArray);
Array<1> B_3(SizeArray);
for(int iter=0; iter < cntmax ; iter++ ){
  for(int i = 2; i < size-3; i++ ){
    A.DataPointer[A.offset(i)] = B_1.expand(i) + B_2.expand(i+1) +
                                 B_3.expand(i-1);

//KCC intermediate C representation for C++ case
auto int iter = 0;
do  {
  auto int i = 2;
  for (; (i < size); i += 1)
     A[i] = B_1[i]) + B_2[1 + i] + B_3[(-1) + i] + B_4[i] + B_5[1 + i];
  iter += 1;
} while (iter < cntmax ); }
```

Figure 7: Test code 3—C++ non-stencil computation

```
//KCC intermediate C representation for Emulated Expression Templates
 auto int iter = 0;
 AA = A.DataPointer;
 offset_0 = A.Stride;
 BB_1 = B_1.DataPointer;
 offset_1 = B_1.Stride;
 BB_2 = B_2.DataPointer;
 offset_2 = B_2.Stride;
 BB_3 = B_3.DataPointer;

do  {
  auto int i = 2;
  for (; (i < size); i += 1)
    AA[i * offset_0] = BB_1[i * offset_1] + BB_2[(1 + i) * offset_2] + BB_3[((-1) + i) * offset_3];
  iter += 1;
} while (iter < cntmax); }
```

Figure 8: Test code 3—EET non-stencil computation

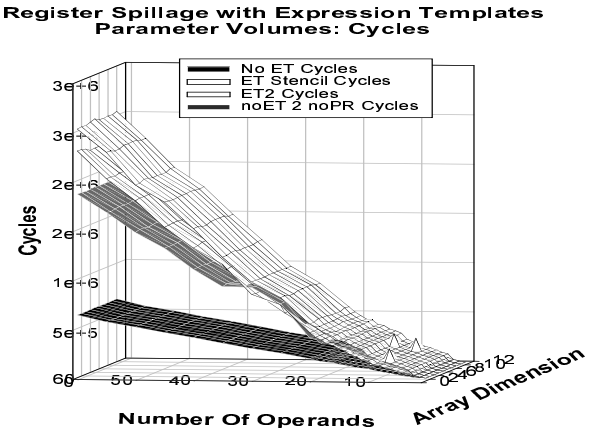
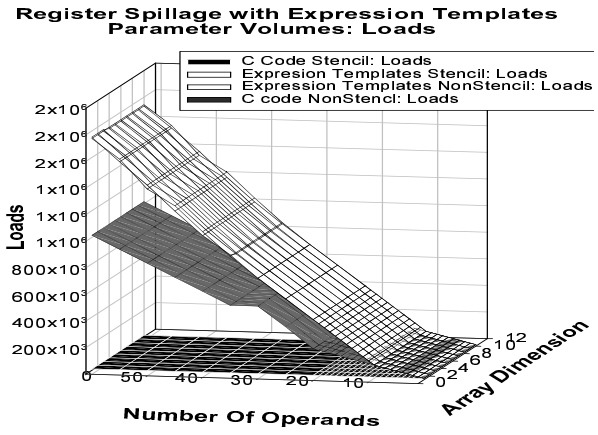
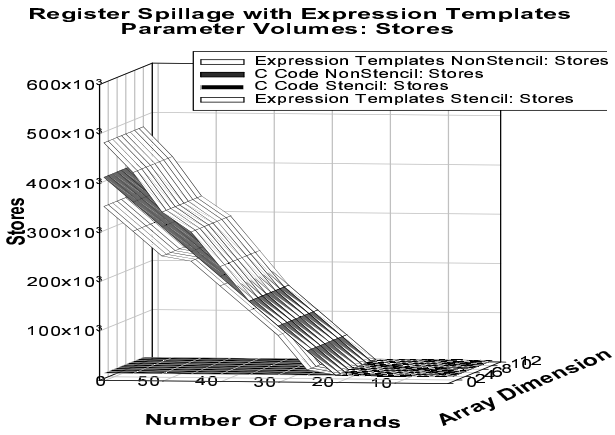


Figure 9: Measured cycles, stencil vs. non-stencil



(a) Memory loads, C++ and EET



(b) Memory stores, C++ and EET

Figure 10: Register Spillage effects, stencil vs. non-stencil code

```
// Emulated Binary Operators code
for( int iter = 0; iter < cntmax ; iter++ ){
  double* restrict T = new double[size];
  assert (T != NULL);
  for( int i = 2; i < size-3; i++ )
      T[i] = B[i] + B[i-1];
  if ( no_fu > 0 )  // disable loop fusion
      no_fu++;
  else
    for(int index = 0; index < size; index++)
      no_fu += T[index];
  for( int i = 2; i < size-3; i++ )
      T[i] += B[i-1];
  if ( no_fu > 0 )  // disable loop fusion
      no_fu++;
  else
    for(int index = 0; index < size; index++)
      no_fu += T[index];
  for( int i = 2; i < size-3; i++ ){
      A[i] = T[i];  }
  delete T;
}
```

Figure 11: Test code 4—Emulation of a 1D 3pt stencil with binary operators

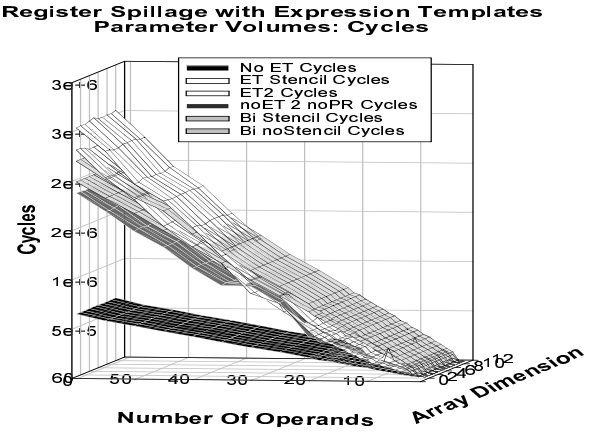
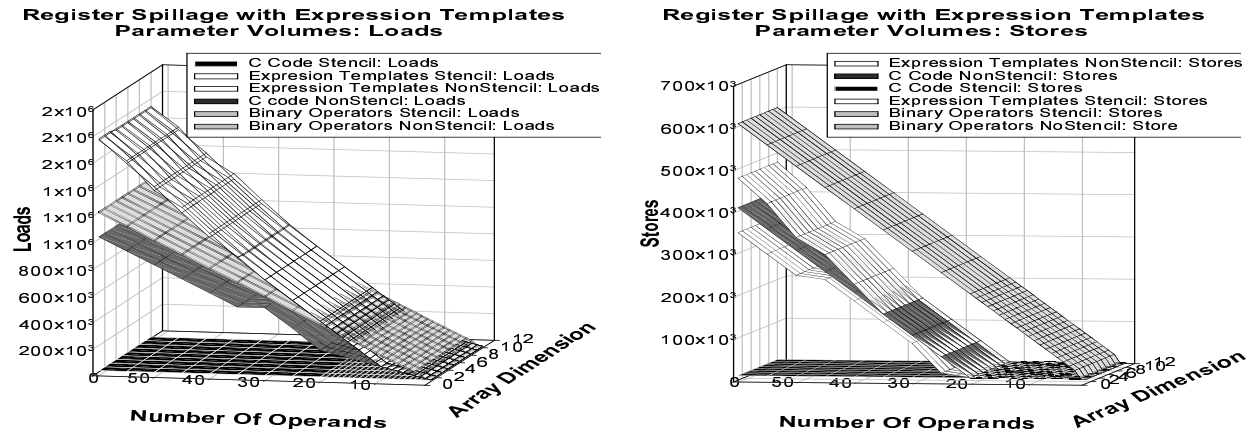


Figure 12: Cycles for C++, EET, and binary operators for stencil and non-stencil code



(a) Loads.

(b) Store.

Figure 13: Register Spillage effects in a non stencil code

```
for( int iter = 0; iter < cntmax ; iter++ ){
  double* restrict T = new double[size];
  assert (T != NULL);
  for( int i = 2; i < size-3; i++ )
      T[i] = B[i] + C[i];
  if ( no_fu > 0 )  // disable loop fusion
      no_fu++;
  else
    for(int index = 0; index < size; index++)
      no_fu += T[index];
  for( int i = 2; i < size-3; i++ )
      T[i] += D[i];
  if ( no_fu > 0 )  // disable loop fusion
      no_fu++;
  else
    for(int index = 0; index < size; index++)
      no_fu += T[index];
  for( int i = 2; i < size-3; i++ ){
      A[i] = T[i];  }
  delete T;
}
```

Figure 14: Test code 5—Non-stencil computation emulating binary operators

number of operands	fixed point registers used	floating point registers used	software pipelining	iterations unrolled
1	5	12	ON	4
2	8	16	ON	2
3	11	20	ON	2
4	12	22	ON	0
5	15	21	ON	0
6	11	17	ON	0
7	12	16	ON	0
8	14	19	ON	0
9	16	20	ON	0
10	15	18	ON	0
11	18	18	ON	0
12	19	17	ON	0
13	19	16	ON	0
14	21	16	ON	0
15	21	17	ON	0
16	21	18	ON	0
17	24	19	ON	0
18	24	20	ON	0
19	24	21	ON	0
20	26	22	ON	0
21	26	23	ON	0
26	26	18	ON	0
31	25	5	ON	0
36			OFF	0

Table 3: Integer and floating point register demand and their impact on pipelining and unrolling

(Quinlan) implementation (as an option in A++) and others'.

As a transformation mechanism ETs are convenient in that the underlying templating mechanism is part of the C++ language, remarkably powerful considering that the templating mechanism was not designed to provide for such complex transformations. On the other hand it is not readily programmable and only addresses single statement optimizations (namely inlining of code and fusing binary operations into a single loop). Stretching the ET mechanism to include more than single statement fusion of binary operations seems to be problematic and forces various compromises which would not be required with more powerful transformation mechanisms. Fundamentally it is macro-like mechanism and cannot be made to perform semantics-based analyses generally needed for non-trivial optimization.

We expect that future work using more powerful source-to-source transformations could result in a superior approach to achieving the desired performance. As such this paper is emphatically not a criticism of the use of object-oriented approaches to scientific numerical software but a recomendation for more work at addressing its optimization through both existing and alternative mechanisms.

References

[1] David Brown, Geoff Chesshire, William Henshaw, and Dan Quinlan. Overture: An object-oriented software system for solving partial differential equations in serial and

parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, March 1997.

[2] S. Goedecker and A. Hoisie. Achieving high performance in numerical computations on risc workstations and parallel systems. June 1997. Mannheim, Germany.

[3] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *International Symposium on Computer Architecture*, June 1997. Denver, CO.

[4] O.Lubeck, Y. Luo, H. Wasserman, and F. Bassetti. Performance evaluation of the sgi origin2000: A memory-centric characterization of lanl asci applications. Submitted to Supercomputing97.

[5] Dan Quinlan and Rebecca Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.

[6] J.V.W. Reynders et al. *POOMA: A Framework for Scientific Simulations on Parallel Architectures*, volume Parallel Programming using C++ by Gregory V. Wilson and Paul Lu, chapter 16, pages 553–594. MIT Press, 1996.

[7] Silicon Graphics Inc. *Performance Tuning for the Origin 2000*, 1997. http://www.sgi.com/Technology/TechPubs/manuals/3000/007-3511-001/html/O2000Tuning.0.html.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[9] Todd Veldhuizen. Blitz++ users manual.

[10] Todd Veldhuizen. Expression templates. Technical Report 5, C++ Report 7, June 1995.

[11] Marco Zagha et al. Performance Analysis Using the MIPS R10000 Performance Counters. In *Supercomputing96*, 1996. http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/INDEX.HTM.