A Generalized Expression Optimization Hook for C++ on High-Performance Architectures

David J. Edelsohn

CRPC-TR94406
February 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

From the *Proceedings of the Scalable High Performance Computing Conference 1994*, Knoxville, Tennessee, May 1994.

A Generalized Expression Optimization Hook for C++ on High-Performance Architectures

David J. Edelsohn
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244

Abstract

C++ has gained broad acceptance as an object-oriented evolutionary extension to the C language, but it severely constrains methods for operating on class objects by forcing all data manipulation through an interface which assumes that all basic operations can be implemented as they are written: as unary or binary operators. C++ allows great flexibility in the creation of complex data structures which can perform the same functionality as built-in types of many other languages, but unfortunately it does not allow an equivalent level of flexibility so that operators acting on those data types can achieve the same level of efficiency as their counterparts in language-level implementations. This limitation becomes even more pronounced on high-performance computers whose advanced features require considerable cooperation between the algorithm, the compiler, and the architecture for maximum performance. This paper describes a language enhancement proposal: a special member function which interacts with overloaded operators as complete expressions.

1 Introduction

The evolving generation of object-oriented languages such as C++ provide a marvelous tool for clearly describing complicated algorithms in simple ways. User-written data classes and operators neatly encapsulate the interfaces for manipulating problem-dependent information as easily as a native language type. Hierarchies of interrelated objects frequently are constructed, such as the now classic examples of drawing a shape on a graphics output device, or arrays of objects manipulated as single, aggregate objects, such as array and matrix class libraries[5, 15, 8], which can provide the functionality described by Golub[10]. The C++ language provides for the creation of complicated, composite objects automatically by implicitly calling the necessary constructors for the object and all of its constituent pieces in the appropriate order, and a symmetric reverse process available to decommission the object.

While all basic operators such as '++', '*' and '%' act in a unary or binary fashion, efficient implementations may require collecting the operations into larger groups. Unfortunately, C++ currently constrains all operators to implementations which correspond to their language context. In the case of aggregate data, this can preclude various optimizations such as vector chaining (pipelining multiple scalar or vector arithmetic operations such as multiply-add) or other special instruction modes in addition to causing the creation of aggregate temporary variables which defeat data caching.[6]

As an example, assume that A, B, C, D and E are conformable arrays. A standard array class library's implementation of an expression such as

```
A -= B*C + ++D/E;
```

would perform a separate loop over all of the elements in each array for each of the five operators involved. Up to five temporary array variables might be created reflecting each of the intermediate results and architectural features such as special add-multiply or vector-chaining instructions could not be utilized. Languages such as Fortran 90[11] and High Performance Fortran[9], both with native array types, can recognize an expression or multiple expressions as a single, related entity which only requires one outer loop, scalar temporaries, and the application of special instruction modes.

C++ operators act upon objects either singly or pairwise. In the case of iterating over the elements of an array, the compiler dispatches to each operator specified in the parse tree in turn, possibly inserting

the code inline instead of as function calls. Inlining removes the function call overhead and allows for some additional optimizations such as common subexpression elimination, but dispatching self-contained operators prevents efficient memory utilization and sequentially operating on elements in pairs of arrays severely taxes memory bandwidth and produces continual cache misses. The distinct arithmetic instructions are not adjacent which disallows further architecture-specific optimization by the compiler such as vector chaining, better register and memory cache utilization, and combining instructions into specific multiple-issue instructions such as multiply-add or Very Long Instruction Word (VLIW) operations.

A considerable amount of effort has been devoted to addressing some of these problems through better policies on the reuse of temporary variables and special optimizations, such as loop jamming, which try to recognize related, adjacent loops and merge them into a single outer loop to provide more opportunities for the other optimizations listed above [4, 13]. C++ makes this very difficult, though, because of the considerable use of pointers and references which hide the effect of many operations and prevent the compiler from making important assumptions allowing code motion and merging. Loop jamming to permit chaining requires C++ to inline loops instead of generating function calls; rearranging the code so that memory allocation and deallocation are segregated from the loops, which itself requires allocation/deallocation with neither side effects nor memory aliases, and then chaining the loops on vector architectures[6]. Unless the compiler merges the loops, chaining on scalar architectures still is not accomplished. C++ provides a programmer with great expressive freedom which leads to many different ways to describe identical functionality — not all of which the compiler can recognize and optimize.

The fundamental problem stems from C++'s bottom-up, hierarchical approach to building objects and methods (functions for accessing and modifying objects). Most compilers internally develop a complete parse tree and basic-block structure which is the essential information necessary to implement deferred expression and/or block evaluation producing the expected performance improvement[1]. One approach to circumvent the language's limitations creates libraries which contain compilers that generate efficient machine code at runtime (Runtime Code Generation or RTCG)[12]. Unfortunately this requires duplicating much of the work performed during the initial compilation phase without having access to additional knowledge in the original source code. Another approach creates specialized class operators which internally optimize multiple operations in their implementation. This, of course, simply provides inelegant, manual optimization in the C++ environment, requiring considerable additional effort and attention from the programmer[6]. Yet another technique which somewhat combines the previous two pre-computes canonical tables of mixed operators up to an arbitrary depth, intercepts expressions at runtime, and maps appropriate combinations of operators to the optimized table[14]. The arbitrarily large and static table can pose significant constraints on this approach.

Deferred expression evaluation implementations of the operators construct a parse tree at runtime instead of directly performing the actual operation[7]. The assignment operation implementation then evaluates the parse tree and performs each of the operations with the additional optimization manually coded into the function using full knowledge of the semantics for the objects' operators. RTCG essentially modifies the semantics of the language to handle a global optimization which cannot be expressed using C++ by implementing portions of a compiler/assembler in the runtime library thereby providing a brute-force opportunity to tailor language enhancements to the program at hand. All of these optimizations fail without this additional effort because C++ does not allow the programmer to adequately describe the implementation of the operators. Instead the programmer must shift the burden to a time when a different but equally flawed view of the problem is available.

2 Language Modification Proposal

The language needs to provide a communication path for use between the programmer and compiler so that the programmer can make best use of the compiler's knowledge and assumptions about the source code and the compiler can best utilize additional hints from the programmer about the intent of the algorithm, objects, and methods. The compiler simply needs to provide a flexible hook at the expression level so that the programmer can describe how to operate on an expression instead of looking at expressions simply as sequences of independent operators.

One would like to utilize the knowledge already available to the compiler to provide this functionality, but C++ must provide a way to describe merging inlined functions — differentiating between the central operations and the details of the implementation. In other words a function to add two array variables

together primarily provides a description of adding elements of an array together and secondarily uses a loop over the elements to implement that functionality. Providing the compiler with a clear distinction between the algorithm and its implementation, i.e. better facilities with which to describe an algorithm, can best solve this limitation.

C++ essentially needs an equivalent to the data object constructor and destructor at the expression level. As mentioned above this can be accomplished at runtime by modifying the operators so that they generate parse tree builders which then are evaluated by the assignment operator. A superior environment should allow the development of operators *in the context of an expression.* A compiler could determine the extent of the expression as usual, possibly merging multiple statements, but then emit code to automatically wrap an operator constructor and destructor around the entire grouping which would provide the necessary initialization and termination for aggregate objects such as arrays requiring iteration. The definition of the data objects, operators, initiators, and terminators all would be defined by the user allowing for complete coordination of private information between class member functions.

It appears sufficient to encapsulate the expression initiator and terminator into a single, user-written function associated with each class. Whenever the compiler would emit code for a statement or expression involving that class, the compiler would substitute the code for the expression member function. The function would receive a key or tag to the as yet unemitted instructions generated by the compiler to implement the expression in the form of a synthetic function pointer, i.e. a pointer to the function generated by the compiler itself representing the final, optimized expression. The programmer can provide any initialization, call the synthetic function pointer as many times as necessary, and then perform any necessary cleanup. The program's control flow does not automatically pass through the function representing the expression; the expression is executed only through the explicit call via the synthetic function pointer. This provides a mechanism for wrapping the operations implementing the expression inside a function which can coordinate its interaction with the class operators. The compiler can generate the code for the expression member function and the expression itself as instructions which explicitly branch to functions emitted elsewhere in the instruction stream or generate code to implement the function calls inline.

A cooperative expression and set of operators clarifies the distinction between the two facets of the implementation problem and completes the symmetry between member data and member functions. If no explicit expression member function exists, the compiler would emit the instructions to implement the expression as usual which is equivalent to an expression member function which evaluates the function pointer once and returns, i.e. "`*func(); return;`". This also maintains compatibility with all current C++ class libraries.

As with all member functions, the expression function provides a member variable, in this case an arbitrary, representative member variable, for determination of specifics about the expression call such as the number of elements in an array over which to loop. Because all of the objects in a expression must be conformable for the expression to have any meaning, the specific member is irrelevant. The choice of object can be left to the compiler as an implementation specific decision.

By allowing the definition of an expression to remain flexible, the compiler has considerable room for optimization. The compiler can choose to invoke the expression function with each pairwise operator in turn which is equivalent to the current approach to discrete operators. Greater efficiency results if the compiler collects entire multi-operator statements or combines multiple statements into basic blocks before generating the synthetic function.

Most operators contain code specific to each operator but not central to the computation, such as checking that two array arguments are conformable. Instead of the expression somehow learning about every argument to every operator, the compiler simply can hoist the code outside of the expression function as constant. This also provides an efficient, central location to place "monitors" and pre- & post-conditions for synchronization of data access.

The determination of invariant code is subtle, so an extension to this proposal allows operators to accept an identical number of additional, optional arguments which correspond to arguments used when calling the function pointer. In other words, the user-written expression member function calls the synthetic function pointer with arguments and these arguments are passed as additional arguments to each and every operator invoked; the operators expect the additional arguments and act upon them accordingly. This allows a direct path for communication between the expression member function and the operators for information such as the iteration index. This clearly delineates active variables in each operator and allows segments of

code not involving those variables, and other variables not defined as volatile, to be moved if the compiler so chooses. Compilers might allow #pragma's to specify sections of code integral to the algorithm and sections providing secondary functionality, which could act as hints to the compiler during code movement optimizations, but the compiler-dependent nature of #pragma's diminishes the benefit. Not moving the invariant code and not calling the expression function with the largest expression only impacts performance, not functionality or results.

Expressions are called in the order that they are referenced. A set of objects composed of other objects all involved in an expression (such as a hash table entry which includes a string) are handled as expressions are encountered. The expression involving the hash entries is implemented and then when the string objects are acted upon, the string expression is called. Hierarchical classes which utilize base class operators are handled from the outside in as well, though this more likely is a question of merging common classes and performing operations in the correct dependence order.

2.1 Example

A traditional matrix class implementation might define the addition operator as follow:

```
Matrix Matrix::operator+ (Matrix m) const {
  EnsureConformance(*this, m);
  Matrix t(rows(), cols());
  for (int i = 0; i < rows(); i++)
    for (int j = 0; j < cols(); j++)
      t(i,j) = elem(i,j) + m.elem(i,j);
  return t;
}
```

Whereas with the use of an expression member function, one would write the above addition operator as follows:

```
Matrix Matrix::operator+
  (Matrix m, int i, int j) const {
  EnsureConformance(*this, m);
  Matrix t(rows(), cols());
  t(i,j) = elem(i,j) + m.elem(i,j);
  return t;
}
void Matrix::expression
  (void *MatrixFunc(int,int)) {
  for (int i = 0; i < rows(); i++)
    for (int j = 0; j < cols(); j++)
      *MatrixFunc(i,j);
  return;
}
```

The stages taken by a conventional optimizing C++ compiler are as follows:

- Parse expression or basic block using class operators.
- Optimize expression.

At this point current C++ compilers simply emit the instructions for each unary or binary class operator inline or as subroutine calls; however, a compiler implementing the proposal described in this paper continues as follows:

- If the function representing the expression is not inlined, emit the instructions representing the expression with an internal label; and when the expression member function (which itself may or may not be inlined) calls the internal expression function using the synthetic function pointer, jump to the internal label.
- If the function representing the expression is inlined, when the expression member function (which also is emitted inline) calls the internal expression function using the synthetic function pointer, emit the internal expression code directly into instruction stream as well.

3 Architecture-Dependent Applications

Allowing object-oriented languages to address entire expressions as single entities instead of implementing each operator in an isolated environment provides benefits for many different types of high-performance architectures. Advanced architectures, such as vector and parallel machines, can achieve dramatic performance increases because complex objects can be implemented with methods which more closely resemble the treatment of builtin types. Simpler processors can benefit from this proposal because this change not only affects the utilization of processor instructions by the compiler optimizer, but improves register and memory utilization as well. In a high-performance implementation, the compiler is expected to optimize the combined expression member function and synthetic

expression function as appropriate for a particular architecture.

Traditional sequential architectures benefit from better memory utilization because temporary values can revert back to their basic scalar types instead of the entire complex object under consideration. The temporaries also can be placed into processor registers more easily because the operator can act on constituent pieces of the object. Data caching naturally receives improvement because all objects within an expression can be active at one time, instead of singly or in pairs, so that one is not repeatedly cycling through the same memory references. A collection of large data objects, such as an array of complex numbers, may not all fit into the cache at one time. Strided accesses also may repeatedly overwrite the cache, not only because of the source array, but because of poorly designed storage for intermediate results.

Expressions producing operations on basic scalar types become even more important to superscalar and VLIW architectures where more than one operational unit can work in parallel. Many architectures have separate addition and multiplication sections in their arithmetic units which can work simultaneously and even feed results directly from one sub-unit to another. Performing all array operations one operator at a time prevents utilization of this feature while expression-level descriptions more easily allow a compiler to move and combine instructions to utilize this capability.

Vector architectures can obtain similar results by strip mining (possibly through explicit coding) the algorithm to utilize vector registers and vector instruction chaining. The compiler will transform the outer loop into a series of chained vector operations because the relationships among operators comprising the expression are clear. Current object classes and methods rarely allow vectorizing compilers to recognize instances where the output of one vector unit can directly feed the input of another vector unit. This severely impacts the performance of, say, a C++ array class library compared to the performance of the Fortran 90 builtin array type.

Distributed memory parallel computers also can benefit by more efficiently managing locality while providing "virtual shared memory" or a "virtual single address space" which is proving very useful for describing parallel algorithms. Problems similar to local memory data caching mentioned above can be avoided by allowing data fetching, including efficient pre-fetching of strided data, to the local system for operation and then updating any remote information. Single messages can be expressed and generated at an appropriate time and without extraordinary efforts to detect opportunities for caching requests and replies.

4 Analysis

This language modification does not provide a complete optimization system for C++ as it does not allow the language to modify the behavior of the compiler. Systems which train the compiler about specific optimization patterns for each library through additional "meta-language" features have a different role to play. Professional, commercial libraries need not be written in C++ and a good case can be made for compiler vendors providing a private or public back-door into the compiler to better handle these cases[2]. This, however, clearly creates a two-tiered system preventing the average user from generating highly efficient libraries without learning about language grammars and optimization patterns described by this essentially new language, assuming that the compiler allows public access to this knowledge.

By providing a hook into code generated for each expression, one can implement many of the same optimizations without straying so far from the original language, e.g. C++. Describing cooperative functions (the expressions member function called by the compiler and the associated operator member functions) provides a much simpler user model while still rebalancing the language by allowing the user to describe the implementation of member functions to manipulate class objects from a higher perspective. This proposal addresses the problem near the end-programmer's regime, where C++ has allowed library development to shift, as opposed to the higher-end vendor supplied library limit. Many different methods along this continuum should be explored and implemented together to give the programmer a variety of choices so that the appropriate solution to fit both the class library and the programmer is available.

The above proposal also can be implemented with multiple passes or source-to-source transformation of the application, such as through Sage++[3]. However this approach greatly lengthens the compilation phase, duplicates much of the work performed by the latter compilation, and possibly removes optimization information which could be utilized by the later compilation. On the other hand this has the substantial advantage of providing the benefits of optimized expressions without requiring modification to all existing compilers (similar to the AT&T Cfront C++ compiler which produces C as its portable output so that it may utilize the local optimizing C compiler).

IBM RISC System/6000 53H	
XL C/6000 1.3.0.8	
0.64	0.36
GCC 2.5.8	
0.91	0.39

DECstation 5000/200	
Mips CC 2.10	
2.61	1.69
GCC 2.3.3	
2.52	1.33

Sun SPARCServer 4/690	
GCC 2.5.6	
1.53	0.72

Table 1: Architecture (System), Compiler, Inner and Outer Loop Duration (seconds).

Table 1 shows the striking effect of loop scope by comparing the same test algorithm performed using "inner" loops — loops for each operator; or "outer" loops — a single loop around the entire expression. A performance increase of between 35% and 57% was obtained depending on the computer architecture and compiler. This variation shows that C++ matrix class libraries must address this efficiency problem to compete as a language for numerically-intensive calculations.

5 Conclusion

C++ cannot address the problem of optimizing expressions without providing a distinction between operators and expressions thereby allowing a cooperative formulation for the problem. Describing operators in isolation does not give compilers for object-oriented languages sufficient information to generate optimal code. One cannot hope that compilers with nearly clairvoyant capabilities can make all of the intervening steps from operators to expressions and blocks without some additional guidance. Explicitly providing access to the expression generation stage of the compiler — the stage where the additional information can produce the most benefit — is the natural choice and an explicit expression function appears to provide the necessary environment.

Both computers with vector processors and with superscalar processors can benefit from improved implementations of vector operations allowing for chaining. Both types of processors can be viewed by an optimizer in a similar way: the limited size vector registers are similar to the limited size data cache. Collecting expressions on aggregate objects such as arrays into the large blocks allows both effective data memory cache management and utilization of vector chaining and multiple-issue instruction modes of operation increasing performance.

Acknowledgments

I would like to thank Dennis Gannon for extremely helpful discussions. This research was supported by an IBM Corporation Graduate Fellowship in Computational Science and IBM Joint-Study Agreement 24680056.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, 1986.

[2] I. Angus. Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More. In *The Object Oriented Numerics Conference*, Sunriver, Oregon, April 25-27 1993.

[3] F. Bodin, P. Beckman, D. Gannon, et al. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. Technical report, Indiana University, 1994.

[4] P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. SUPERB Support for Irregular Scientific Computation. In *Proceedings of the Scalable High Performance Computing Conference*, pages 314–321. IEEE Computer Society Press, April 1992.

[5] K.G. Budge. PHYSLIB: A C++ Tensor Class Library. Technical Report SAND91-1752, Sandia National Laboratory, Sandia, New Mexico, 1991.

[6] K.G. Budge, J.S. Peery, and A.C. Robinson. High-Performance Scientific Computing Using C++. In *USENIX C++ Technical Conference Proceedings*, pages 131–150, Portland, Oregon, August 1992. USENIX Association.

[7] R.B. Davies. Notes for the Library Working Group of WG21/X3J16. Presented at C++ Standards Committee Meeting, March 1991.

[8] J.J. Dongarra, R. Pozo, and D. Walker. Lapack++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra. Technical report, Oak Ridge National Laboratory, 1993.

[9] High Performance Fortran Forum. HPF Language Specification. Technical Report CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, May 1993.

[10] G.H. Golub and C.F. Van Loan. *Matrix Computations, 2nd ed.* Johns Hopkins Press, Baltimore, 1989.

[11] ISO/IEC 1539:1991(E) and ANSI X3.198-1992.

[12] D. Keppel, S.J. Eggers, and R.R. Henry. A Case for Runtime Code Generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, Seattle, November 1991.

[13] C. Polychronopoulos, M. Girkar, M. Haghighat, et al. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 423–453. MIT Press, 1990.

[14] D. Quinlan and R. Parsons. A++/P++ Array Classes for Architecture Independent Finite Difference Computations. In *The Object Oriented Numerics Conference*, Sunriver, Oregon, April 1994.

[15] Dyad Software. *M++ Matrix Class Library Reference Manual.* 1991.