Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More

IAN G. ANGUS
Boeing Computer Services, Seattle, WA 98124-0346

ABSTRACT

So far C++ has made few inroads into the realm of scientific computing, which is still largely dominated by Fortran. Of the few attempts that have been made to apply C++ to numerically intensive codes, the results have often suffered from severe performance problems. A careful examination of these problems indicates that they are unlikely to be solved by incremental improvements in compiler optimization technology. The flow of this article will: motivate the discussion by describing a common efficiency problem that arises when numerical codes are programmed in C++; discuss some potential solution strategies that we believe are viable in the near term, but not over the long term; introduce a mechanism by which a compiler can load domain-specific and class-specific optimizations on an as needed basis. A simple interface that will enable this feature will be presented. Although our immediate motivation is that of numerically intensive codes, our approach is applicable to all application domains.

1 INTRODUCTION

There are programming communities in which C++ has had little or no impact. One such community consists of researchers and programmers who write numerical codes, typically in Fortran, for use in applications such as computational fluid dynamics (CFD). Many of these applications are built on a succinct mathematical structure that suggests that these applications would be good candidates for being programmed in C++.

To achieve greater performance a few of these applications are now being migrated to distributed memory parallel computers. The complexity and cost of this task have caused some researchers [1–7] to investigate whether coding applications in C++ would make the management of parallelism (and the applications) easier. All have been favorably impressed with the software engineering aspects of C++, and they unanimously believe it did help to manage the parallelism. Others [4, 8–10] have investigated the applicability of C++ to scientific computing, also with encouraging results.

Unfortunately, in many cases performance problems of varying difficulty have been encountered with the C++ implementations [1–4, 11]. Because the nature of these applications pushes the envelope of what can be computed, the observed degradations in performance (which are modest by many standards) are often deemed to be unacceptable.

Received April 1993
Revised June 1993

Scientific Programming, Vol. 2, pp. 123–131 (1993)
CCC 1058-9244/94/040123-09

2 SCIENTIFIC APPLICATIONS

The author's experience is derived from a CFD application that solved for the fluid flow around an aircraft [1, 11] and an image processing application [5]. These problems can be characterized as having relatively few large objects. Large in this context means each object contains upward of tens of thousands of floating point values.

For the purpose of this article we will use the humble vector class (stripped of all detail that we do not need and depicted in Fig. 1) as a familiar example. We ask the reader to remember that the vector is a greatly simplified representation of the structures that are used in real CFD codes. This example is meant to be instructive, and not representative: it omits many of the complexities of real numerical applications.

3 RELATIVE SPACE/TIME EFFICIENCY

The observed performance degradations exhibit the following characteristics:

1. They are nonlocal, i.e., using the usual performance monitoring tools such as *prof* does not give much insight into what is happening. The benchmarking of individual components of the code may not reveal the potential for problems.
2. The timing differences between C++ and C are strongly machine dependent. The C++ to C ratio ranged from 1.5 at best, to 3 and worse [11].
3. The relative memory consumption of C++ code is often greater than a C code with the equivalent functionality. For the CFD code described [1], we estimate that the excess memory consumption was about a factor of 50%.

```
class Vector {
    public:
        Vector();
        Vector& operator=(const Vector&);
        Vector& operator+=(const Vector&);

        friend Vector operator+(const Vector&,const Vector&);
};

{
Vector  A, B, C, D;
        // ...
A = B + C + D;          // Fragment 1
        // ...
A = B; A += C; A += D; // Fragment 2
        // ...
}
        // Fragment 3
        // Functionally equivalent C version
        // which violates C++ encapsulation!
{
for ( int i = 0 ; i < N ; i++ ) {       // N = Vector size
        A[i] = B[i] + C[i] + D[i];
        }
}
```

FIGURE 1 C++ vector code fragments.

For many of the numerically intensive problems, inefficiency in memory usage can be as, or more, damaging than an execution time penalty.

3.1 The Usual C++ Suspects?

Before pointing fingers, consider the possible role of the usual suspects.

1. Virtual functions: The CFD code [1] made very little use of C++ inheritance—it just did not need it. Optimizing virtual dispatch, and methods such as customization [12] will not help.
2. Memory allocation: The objects we found useful contained very large amounts of data. The overhead of allocating the memory was negligible compared to the work done in a typical method.

These issues must not be ignored; however, they are not the only sources of trouble and were not relevant to our applications.

3.2 Source of the Numerical Efficiency Problem

The inefficient use of memory occurs because the mathematical abstractions force the use of some temporary variables that are very big [11]. The inefficiencies in time occur because of the memory usage, and because the abstraction boundaries force a particular access pattern on the data. This pattern happens to be quite different from that of a C or Fortran code of similar functionality [2, 11].

Effectively, the pattern of traversal precludes effective use of either the cache (the objects are too big to fit into it) or the registers. Hence, the code runs at memory access speed. Depending on the hardware used; this may or may not be a problem. It could be a disaster.

The types of optimizations that the C++ compiler has precluded (compared with C and Fortran) are those that act by grouping terms from

different statements or parts of an individual expression. In Fortran and C codes the program is (often) written so as to enable the compiler to exploit the registers and cache effectively. The result of these optimizations being implemented directly by the programmer is that the program often becomes harder to understand. One of the reasons for using C++ in the first place is to separate concerns of the problem the program solves from details of implementation (and optimization).

4 SOLUTIONS?

The reason for the poor efficiency of C++ relative to the competing languages is that the abstractions, which were so useful in aiding the writing of an application, hamper the optimizer. The process of optimizing code proceeds by the compiler recognizing enough of the semantics of the code to be able to apply transformations (the optimizations) that are known to produce a resultant code that is mathematically equivalent to the original.

For the case of C++ the optimization process is thwarted by the presence of user-defined data types for which the compiler has no semantic understanding. The compiler's *only* option is to use function inlining in an attempt to reduce the code to a set of (lower level) abstractions that it might understand.

4.1 Must We go Beyond Inlining Functions?

It is valid to ask if our efficiency problems were just a consequence of function inlining (and the related process of interprocedural analysis) mechanisms being rather primitive. Although we would be happy to see greatly improved capabilities in this area we are skeptical that it is a long-term solution.

In Figure 1 we depict a tiny portion of a vector class. The two operators shown are presumed to be implemented with semantics analogous to integer operations. The two code fragments represent different performance characteristics. The first uses more memory (because of temporaries) than the second, but suffers less loop overhead.

For maximum efficiency we would like to transform the first code fragment into the form of the second fragment, unfold the operations, and fuse the loops to deliver the equivalent of the third fragment. Note that the third fragment violates encapsulation and so cannot be effected by a C++ programmer.

Although this approach may be theoretically possible, it is beyond the current state of the art (as witnessed by the absence of any C++ compilers that could effect the transformations above - today). Hence, other possibilities will be considered.

4.2 Short-Term Solutions

To retain the expressive syntax, there are two strategies that can be fruitfully applied in the short term:

1. Design libraries that employ lazy evaluation and method combination [13]. Effectively, the vector operations of Figure 1 are not actually executed as they appear; instead the library behaves as though it is a specialized compiler. The library builds a parse tree that it will attempt to evaluate. This approach appears promising for some specialized codes. If it works, it can be implemented now.
2. Hard code the knowledge of certain critical classes into the compiler. The critical question is: Which classes do I build in to the compiler? Even if agreement could be reached, this solution is only viable for compiler vendors, those who have a lot of leverage with their compiler vendor, or those who have both the access and the expertise to modify a compiler themselves.

If we can achieve the desired efficiencies via clever implementations of libraries then the library approach is preferable. Deferred evaluation is an appealing strategy; hence, we will devote some time to pointing out its strengths and weaknesses vis a vis a compiler.

1. Building and compiling a parse tree at runtime will not be cheap in time and possibly memory. Hence, we would like to amortize this effort over multiple invocations of the same code. This requires library to tag where the tree was called from, i.e., some external context must be captured at runtime. The method used to do this is unlikely to be portable.
2. For short vectors, the library would need to use a more "traditional approach" to eliminate the cost of the runtime compiling. The

compiler does not suffer from this problem, it will be able to generate optimal code (of the same structure) for all sizes of vector. (We could demand that the user use different flavors of vector to denote this difference—we reject that approach).

Note that these problems are all specific to the vector (and similar) example.

There are disadvantages with putting these optimizations in the compiler as well; however, they will be discussed later once our proposal has been introduced.

4.3 Pragma

One approach to communicating optimizations to the compiler is with *pragmas*. *Pragma* suffers from one serious deficiency: It amounts to a compiler and library implementation-dependent extension to the language. Our thesis is that if it is compiler or library implementation dependent, the programmer should not see it. In addition, for pragmas to be of general use they need to be standardized, otherwise chaos will eventually rule.

4.4 Class Optimization Specifications

Another approach would be to extend the C++ language to allow legal optimizations to be specified as a part of the class definition. Aside from our natural aversion to extending the language, we believe that this approach is flawed for two reasons:

1. No language general enough to specify all possible optimizations is known. Such a language may be impossible to design!
2. The transformations with the biggest payoff (at least for vectors) violate the abstraction boundaries imposed by the C++ classes. These optimizations are dependent on implementation knowledge which we do *not* want to embed in the class body.

We take the point of view that optimization is a matter of implementation and not interface.

5 A GENERAL SOLUTION

The methods described in Section 4.2 represent extremes; leave the compiler alone, or, put everything in the compiler by embedding specific libraries inside the compiler. We will propose a solution that subsumes the second approach, is much more flexible, and does not foreclose upon the first.

The critical problem is that the number of user-defined data types is potentially unbounded, so we need to be able to handle a potentially unbounded number of useful transformations.

The recommended solution is as follows:

1. For each class type that demands special optimizations, encapsulate the allowed transformations (defined as transformations on the compiler's parse tree) inside an "optimization module."
2. Define an interface so that the compiler can dynamically link the optimization modules when needed. During compilation when the compiler encounters the use (rather than just the definition) of a class object it searches a database for any optimization modules that correspond to that class. If found, they are dynamically linked into the compiler.
3. The compiler then proceeds to apply the loaded transformations against the internal representation of the user's program.

6 WHAT MUST WE ADD TO THE COMPILER?

Figure 2 shows the basic phases of the compiler/linker with the components we are proposing to add (shown as shaded). The link phase is included for completeness. The basic "components" of a library (include headers and binary code) are also shown along with two components that we will introduce. The phases of the compiler/linker that access the library are also denoted.

There are a number of basic architectural features that deserve mention:

1. The "new compiler phase" is not activated until *after* the program has been read in and completely typechecked.
2. The class-specific optimizations are then applied *before* standard C++ optimizations such as function inlining and return value optimization are applied. We postpone function inlining because we want to attempt to apply the higher level (we hope) class-specific optimizations first. We expect to iterating with these steps alternating, and Figure 2 denotes this.

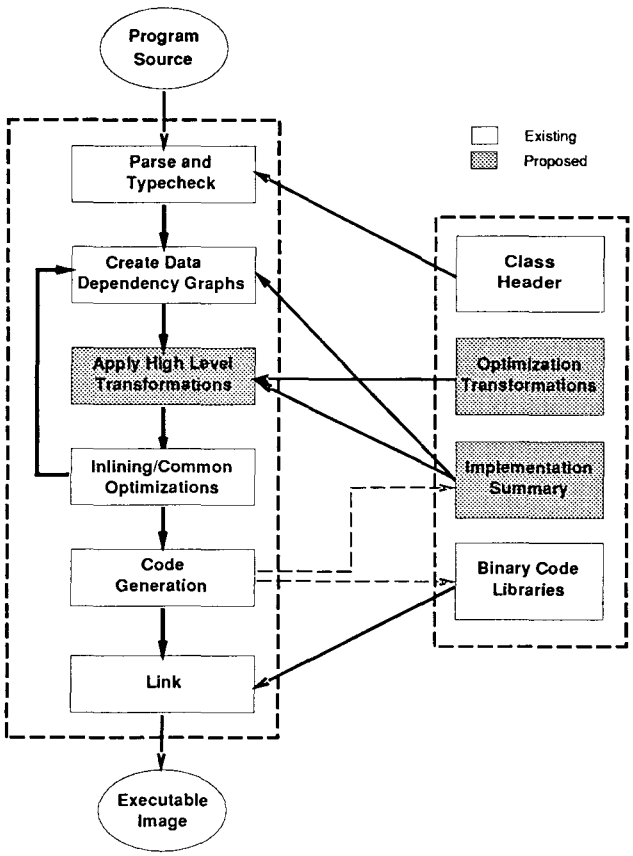

FIGURE 2 Compiler architecture.

3. Our approach is reliant upon *local* analysis. This assumption is to allow for tractability. In addition, we want to retain the separate compilation model to the same degree that C++ admits this model.

7 DESIGN AND IMPLEMENTATION APPROACH

Our approach (the compiler itself being written in C++) follows the structure of Dewhurst [14]. To reduce the implementation effort a first implementation would transform the user's C++ code to a new C++ program that has had the optimization module transformations applied to it. An implementation at this level would be sufficient to test the concept.

The important features are:

1. The type lattice implemented within the compiler has an interface that is exported to the outside world. Part of this interface for the (C++ class) "ClassType" nodes is shown in Figure 3; we will ignore the (few) optimizations that can be applied to all class types for this discussion.
2. The author of an optimization module inherits from this interface as is shown in Figure 3. The optimizer module defines a version of the *Optimize* function. Through this function, which is applied to an expression (which could be a list of statements), the class-specific transformations are applied (see Fig. 4). We have assumed here that whatever language is used to write the optimization module, it can be "compiled" to C++. Note that the transformations may involve more than one class type even though one particular class is used to label the module.
3. This use of inheritance enables the compiler to dynamically link [15] the optimization module and known that it is type-safe with respect to its internals.
4. The space of all transformations that could be applied to an expression is determined by the type of that expression, or the "nearest" nonvoid type if such a type can be found.

7.1 Form of the Optimization Modules

Typically C++ has been used to write both the libraries and the user's application; however, this

```
class Expr {     // Expression node
        // ...
    protected:
        const Type      *type_p;
    };

class ClassType : public ... {
    public:
        virtual Expr *Optimize(Expr *e_p)
                { return e_p; } // Do nothing

    };

class Vector_Type : public ClassType {
        // ...
    public:
        virtual Expr *Optimize(Expr*);
    };

Expr *Vector_Type::Optimize(Expr *e_p)
{
Expr    *new_e_p;
    // Implement optimizations that are specific
    // to vectors, build a new expression
    // tree rooted with new_e_p
return new_e_p;
}
```

FIGURE 3 Class type and vector_type C++ interface.

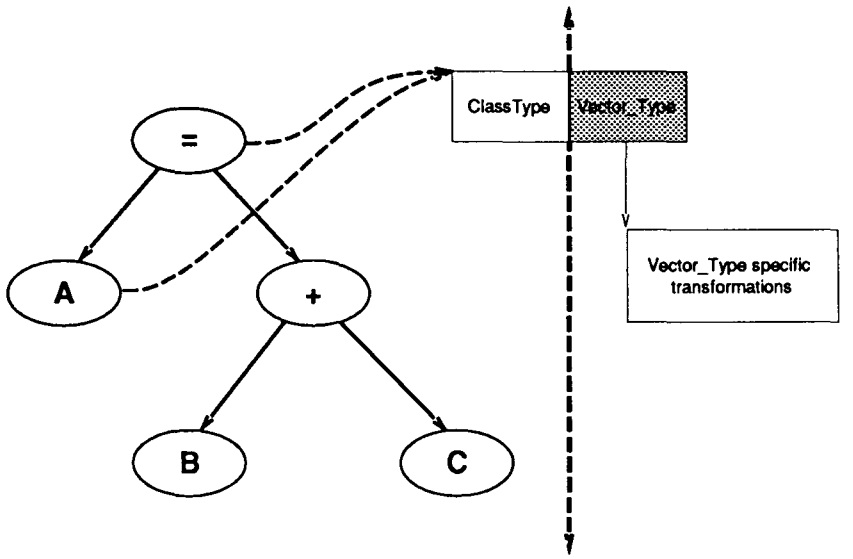

FIGURE 4 The expression tree.

is not appropriate for the part of a library that is implemented as an optimization module. Optimization modules can be thought of as libraries that are loaded into the *compiler;* rather than being loaded by the *linker* (as for traditional libraries). The contents of an optimization module would be a set of patterns that we hope to match and associated transformations that would be applied whenever a match is found. We expect that some form of "tree transformation" language would be used instead of C++, and that this code would be compiled to C++ to permit dynamic linking to work.

7.2 Choice of Transformations

In this scheme, envisage the class-specific optimizations being applied to sequences of operations. The operations are individual invocations of operators or functions and not statements. The base class for these nodes is represented as *Expr* in Figure 3. In real applications we should expect operations, as written, to be presented in an order that could confuse an optimizer using this grouping strategy. The simplest example would be that of two unrelated statements being in the reverse order. To solve this problem we propose that the compiler do sufficient dependency analysis to effectively group operators by data dependency.

We will then apply the transformations following the branches of the data dependency graph.

7.2.1 Data Dependency Analysis

The optimization approach proposed requires that expressions that can be profitably grouped have to be adjacent so that the optimization functions will recognize the groupings. As a concrete example, the code of Figure 5 would occur if we simply inline expanded one level of abstraction of the numerical interpolation and divergence operators such as existed in Angus and Thompkins [1]. The important point of this example is that the resulting ordering of the operations is the least efficient in memory consumption. If we could interchange the second and third statements we could reduce the memory consumption considerably. We know that we can interchange Expressions 2 and 3, but does the compiler? In general the answer is no. To enable the compiler to effect this interchange, it has to know that there are *no* data dependencies relating the two functions. The compiler will have to deduce this because there is no language feature for describing all potential side effects to the caller of a function. We require that the compiler does a better job of dependency

```
average_flux_x = average_x(flux);          // 1
average_flux_y = average_y(flux);          // 2

delta_density  = grid->div_x(average_flux_x);// 3
delta_density -+= grid->div_y(average_flux_y);// 4
```

FIGURE 5 C++ code fragments.

analysis than just assuming that functions can depend on everything. We need a facility that will propagate this information (which is a summary of what might be obtained via interprocedural analysis) to the caller and the compiler. This facility is called the *implementation* interface.

7.2.2 The Implementation Interface

The implementation interface is a summary of the implementation details of the function. It is a version of interprocedural analysis. The summary information for a function is generated either by the compiler when the function is compiled, or by the hand of the author of the function. The summary information is passed surreptitiously by the environment to the compiler for use when this function is called.

This interface allows the compiler to import a description of all side effects the function could cause. A first implementation of this would include (at least) a list of all external variables that the function accesses, and whether those accesses are reads, writes, or both. For the purpose of this discussion all we need is more refined information that enables us to eliminate (most) false dependency constraints.

With this information the compiler is able to determine whether or not statement 3 (in Fig. 5) has any dependencies upon statement 2.

There are a number of problems with this approach*:

1. If the function has not yet been compiled, then the worst has to be assumed. We do not believe this to be bad because we view these functions as being existing library functions.
2. A program that rearranged code is vulnerable if the assumptions under which the client code are violated, for example, a new version of the library implements a new version with a wider set of side effects.

There is no guaranteed approach to preventing this kind of error. The only defense is checking of version information of the hidden interface, and care on the part of the library programmer that he or she has not expanded the effect of the function. A tool to do this at this level could easily be built.

This is one area in which the library's only approach has an advantage. Provided that *no* assumptions about how to write efficient code are propagated to the user, the library writer is solely responsible.

* Note that checking the correctness of template usage invokes a similar set of concerns, and perhaps solutions.

7.2.3 Application of the Optimizations

Once the best data dependency graph has been created, the type-based optimizations are applied by grouping statements along the edges of the graph. This process starts from the node of the graph that represents the function entry point. The stopping condition for applying some optimizations will be class specific.

7.2.4 Ambiguous Cases

A simple case that the approach up to now would be confused by would be two or more statements that were completely independent. A simple example is shown in Figure 6. In this case, there are multiple branches emanating from the root node. In this case we would propose that the dependency graph be augmented with a graph that tends to group items of the same type together. This allows for our optimizations being selected based on types.

The detection criterion here is that given a starting node, there are two (or more) branches from the graph that have the same types used along both arms until those paths merge (which they must when the function returns. The compiler would group together, as a single branch, the branches that have the same (or related by inheritance) types.

7.2.5 Gravity

One interesting case would be to define message-passing function calls as a pair consisting of a "request" and a "request completed" message. It might be advantageous to push the request as early as possible, and the corresponding check on completion as late as possible. We have not de-

```
Vector   A, B, C, D, E, F;
int      i;

         ...

A = B + C;
i = 10;
D = E + F;
```

FIGURE 6 Independent statements.

vised a mechanism to implement this, but we expect it to be possible. In essence, we envisage a gravity-like mechanism, which causes functions to want to be executed as early, or as late, as possible.

7.3 Advantages

This scheme has several notable advantages:

1. The C++ language is not modified in any way. In particular, on one platform a class library could be implemented in the usual manner, while on another platform, the optimizer modules would be used exclusively.
2. The compiler is provided easy access to the high level transformations such as were described in Section 4.1.
3. That the use of these transformations might depend on the target machine will not be reflected in the programmer's application code.
4. The possibility of configuring the optimization process is greatly enhanced. A configuration system can now be envisaged, analogous to the X-resource database, that would be equivalent to setting compiler switches at the granularity of individual classes, or groups of classes. This would give fine grain control over the speed-memory demands of particular classes depending on the context in which they were used.
5. Library designers will now have access to the compiler in ways that they do not have today. In particular, the designers of libraries would know that performance-critical application domain-specific transformations could be supported independent of how cooperative their compiler vendor is. They will always provide a set of libraries so as to cover the case of compilers that do not support optimization modules.

7.4 Potential Disadvantages

At this stage one can only speculate about all of the things that might undermine the utility of this approach. Possible problems include:

1. If the compiler has loaded multiple optimization modules the likelihood of conflicts between the applicable transformations is high. Some mechanism will be needed to deal with this.
2. The applications considered did not make much use of inheritance and polymorphism. The applicability of the optimization scheme presented to code that uses these features heavily is unknown.
3. There may be some transformations that cannot be effected within the confines of the interface introduced here.
4. This approach allows for executable code to be loaded into the address space of the compiler. Assignment of blame for compiler crashes and/or the generation of incorrect code will likely be difficult.
5. The time is takes the compiler to search for good (rather than the absolute best) transformations may be prohibitive.

7.5 Is this a Language Extension?

The mechanism described could be used to extend the C++ language in ways that are unpredictable. Even worse, it could be used to silently change the meaning of a program. If used well, this should not be a problem. In particular, the intent of this tool is to do nothing more than what optimizers do already—apply mathematically correct transformations that do not alter the meaning of the program. Unfortunately, we do not believe that we could enforce correct use.

7.6 Who Would use this Capability?

Our approach presumes that applications programmers will never attempt to use this facility; rather, it will only be used by library designers. We are making some severe assumptions about the capability of the programmers who would actually use this compiler "back door."

7.7 Environmental Requirements

Our approach does not rely upon environment support above or beyond that which C++ requires. At the most basic level, it does not need to be any more sophisticated than the search paths that are currently used by the C preprocessor.

It has been assumed that the build procedures for this environment are slightly better than would typically be implemented with *make*. This is to ensure that the hidden interfaces that might be generated by the compiler would be up to date

with respect to all clients. Sadly, this restriction forbids the use of this scheme with any form of recursion. Recursion has not been a factor in scientific contexts that we are familiar with; however, somewhere it will cause somebody some trouble.

As a final point, nothing in this proposal requires source code to be shipped rather than binary code representations of libraries.

8 COMPARISON WITH OTHER WORK

None of the individual ideas expressed here are particularly new; however, we believe that the composition of them within a C++ compiler is.

We note that many compilers today recognize certain names (which are otherwise just undistinguished identifiers) and then generate special code for them. Gcc uses modules (compiled into gcc) to specify peephole optimizations. Our approach is largely made possible by the ability of C++ to link in derived classes (the optimization modules) in a type safe fashion [15], while the compiler is executing.

Work on program transformation system [16] is also relevant to the discussion here—in particular to the way in which the optimization modules may be programmed. In addition interesting techniques have been developed for optimizing object-oriented languages other than C++ [17].

9 FUTURE DIRECTIONS

The final goal of this work will be to define an interface between the compiler proper and the optimization modules. Should this prove successful, the next step would likely be the development of "little languages" that facilitate the succinct specification of allowed transformations that would then be compiled into optimization modules.

10 CONCLUSION

There are applications for which the efficiency of the executable derived from C++ code can be unacceptable.

We have described a method by which groups of potential optimizations may be designed, implemented, and archived external to the compiler. When required, the compiler dynamically loads the needed optimizations through a type-safe interface so as to make the transformations available.

The gains will be twofold; it will be possible now for C++ compilers to invoke transformations powerful enough to make it competitive with Fortran and C, and domain-specific optimizations could now be developed in tandem with the libraries and independent of the compiler, and the C++ language itself.

REFERENCES

[1] I. G. Angus and W. T. Thompkins, *Fourth Conference on Hypercubes, Concurrent Computers, and Applications.* Monterey, CA, 1989.

[2] D. W. Forslund, et al., *USENIX C++ Conference.* San Francisco: USENIX, 1990.

[3] A. C. Robinson, K. G. Budge, and J. S. Peery, *USENIX C++ Conference.* Portland: USENIX, 1992.

[4] T. Wicks, D. Curtis, and K. Pennick, *An Assessment of the Suitability of C++ for Finite Element Class Design.* Technical Report, Boeing Computer Services, 1991.

[5] I. G. Angus, *Scalable High Performance Computing Conference.* Williamsburg, VA, 1992.

[6] S. Bhatt et al., *Scalable High Performance Computing Conference.* Williamsburg, VA, 1992.

[7] D. Quinlan, D. Balsara, and M. Lemke, AMR++, A C++ *Object Oriented Class Library for Parallel Adaptive Mesh Refinement Applications.* 1992.

[8] T. J. Ross, et al., *Computing in Civil Engineering.* Dallas, 1992.

[9] T. Keffer, "Object oriented numerics, Part 1: Vectors, matrices, and all that stuff, *C++ J.*, vol. 1, 1991.

[10] T. Keffer, "Object oriented numerics, Part 2: virtual algorithms," *C++ J.*, vol. 2, 1992.

[11] I. G. Angus and Janice L. Stolzy, *C++ at Work Conference.* San Jose, 1991.

[12] D. Lea, *USENIX C++ Conference.* San Francisco: USENIX, 1990.

[13] R. B. Davies, *The Newmat Class Library* (available by ftp). 1992.

[14] S. C. Dewhurst, *USENIX C++ Workshop.* USENIX, 1987.

[15] J. E. Shopiro, S. M. Dorward, and R. Sethi, *USENIX C++ Conference.* San Francisco: USENIX, 1990.

[16] D. R. Smith, "KIDS - a semi automatic program development system," *IEEE Transac. Software Eng. Special Issue Formal Methods Software Eng.*, 1990.

[17] D. Ungar et al., "Object, message and performance: How they coexist in self," *IEEE Comput.*, 1992.