Parallel Object-Oriented Framework Optimization *

Dan Quinlan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

Abstract

Object-oriented libraries arise naturally from the increasing complexity of developing related scientific applications. The optimization of the use of libraries within scientific applications is one of many high-performance optimizations, and is the subject of this paper. This type of optimization can have significant potential because it can either reduce the overhead of calls to a library, specialize the library calls given the context of their use within the application, or use the semantics of the library calls to locally rewrite sections of the application. This type of optimization is only now becoming an active area of research. The optimization of the use of libraries within scientific applications is particularly attractive because it maps to the extensive use of libraries within numerous large existing scientific applications sharing common problem domains. This paper presents an approach toward the optimization of parallel object-oriented libraries.

ROSE[1] is a tool for building source-to-source preprocessors, ROSETTA is a tool for defining the grammars used within ROSE. The definition of the grammars directly determines what can be recognized at compile time. ROSETTA permits grammars to be automatically generated which are specific to the identification of abstractions introduced within object-oriented libraries. Thus the semantics of complex abstractions defined outside of the C++ language can be leveraged at compile time to introduce library specific optimizations. The details of the optimizations performed are not a part of this paper and are up to the library developer to define using ROSETTA and ROSE to build such an optimizing preprocessor. Within performance optimizations, if they are to be automated, the problems of automatically locating where such optimizations can be done are significant and most often overlooked. Note that a novel part of this work is the degree of automation. Thus library developers can be expected to be able to build their own specialized compilers with a minimal compiler background. The resulting compilers don't extend the C++ language, but only extend the compiler's ability to recognize and leverage the use of user-defined library abstractions within an application to perform optimizations.

For completeness, an example optimizing preprocessor for a parallel array class library is included to demonstrate the complete use of ROSETTA and ROSE to build an optimizing preprocessor. These results combine the use of the recognition techniques presented in this paper with those of a preprocessor-based transformation approach. The specification of transformations and the details of the construction of full preprocessors is outside the scope of this short paper, however some details of the compiler infrastructure we are using can be found in ROSE[1].

1 Introduction

To application programmers the use of a library to provide new abstractions might appear to provide a language extension specific to the application domain

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

targeted by the library's designer. With an object-oriented language the abstractions provided within the library can be endowed with significant syntactic sugar (function overloading) so as to make them largely indistinguishable from an additional language feature (such as a new type). Such object-oriented libraries are however **not** extensions of the language for one essential reason; the C++ compiler does not recognize or optimize the library's abstractions. The reason for this is that there is no mechanism to communicate the library's abstractions to the typical C++ compiler. Thus no mechanism exists to introduce optimizations that are specific to a library's abstraction. A C++ language compilation approach that would permit library writers to communicate the optimizations associated with the abstractions within their libraries would complete the essential step in permitting object-oriented libraries to be considered as equivalent to language extensions (or would at least muddy the water). This paper presents an essential piece of this work to open up the development of C++ compilers so as to permit object-oriented library/framework developers (instead of only compiler writers) to build portable and easily maintained compilers that are capable of optimizing the abstractions represented by their libraries. We believe that this work is a critical part of future performance optimization for object-oriented libraries.

We define a mechanism to build preprocessors to automate the optimization of applications containing user-defined abstractions via source-to-source transformations. Clearly not all optimizations are appropriate for introduction via source-to-source transformation, but such an approach is intended to be complementary to a vendor's C++ compiler, which is relied upon for all lower level optimizations. This paper will present a powerful mechanism to represent a critical phase of that work; automatically recognizing the use of complex object-oriented abstractions at compile-time. Our approach extends well beyond the tedious limits of pattern matching and automates the construction of whole grammars and parsers to re-represent the program's abstract syntax tree (AST) within the compiler. The resulting ASTs using the generated grammars are dramatically simplified since they explicitly identify language elements (expressions and statements) specific to the user defined object-oriented abstractions. Typically such object-oriented abstractions are made available in object-oriented libraries or frameworks, so in this way our approach is well suited to the optimization of applications using such libraries.

The following sections in the paper detail **ROSETTA**'s use within **ROSE**, its implementation and how it leverages existing projects particularly the EDG C++ front-end and a modified version of the SAGE II source code restructuring tool. In further sections we describe some of the important features. We present some performance results from the use of this recognition approach within **ROSE** and finish with conclusions about its use.

2 A Motivating Example

To make the discussion within the paper as concrete and easily understood as possible, we will use a motivating example from the A++/P++ array class library[13] and define our grammars to optimize this example. **ROSETTA**, and **ROSE**, are not at all specific to this or any other specific example. However, both **ROSETTA**, and **ROSE**, are being used to optimize the performance of the A++/P++ array class libraries.

Within our motivating example we consider the following trivial five-point stencil array operation:

```
floatArray A(100,100);
floatArray B(100,100);
Range I(1,98),J(1,98);
A(I,J) = B(I-1,J)+B(I+1,J)+B(I,J)+B(I,J+1)+B(I,J+1);
```

In this code fragment, `A` and `B` are multidimensional array objects, `floatArray`. A++ is a serial array class library, P++ is a parallel array class; data in the arrays are distributed across multiple processors if P++ is used and communication is automatically introduced. Both the A++ and P++ libraries share an identical interface, so that the same code can be developed using A++ and recompiled and executed in parallel using P++. In this example, `I` and `J` are `Range` objects that together specify an two dimensional index space of the arrays `A` and `B`.

3 ROSETTA

ROSETTA is a tool developed for the manipulation and construction of grammars. It permits a C++ *Meta-program* to be defined which, when executed, builds tools like Sage II [8] from the user's manipulation of the C++ grammar (using the ROSETTA object-oriented library). Specifically, elements of SAGE II source code form the definition of the C++ grammar's implementation within **ROSETTA**. **ROSETTA** is not specific to C++ in any way, but is used currently for the development of the C++ grammar and higher level grammars that include user defined types, statements, expressions, etc. It is **not** a novel part of this work to have defined a mechanism to generate SAGE II, modified or not. The novel aspect of this research is that higher-level grammars can be automatically generated in *addition* to the modified SAGE II. This paper presents **ROSETTA** as the mechanism by which critical parts of a final preprocessor are customized for a framework's abstractions; and automatically generated. Some aspects of the infrastructure for building the final preprocessor are presented in ROSE [1].

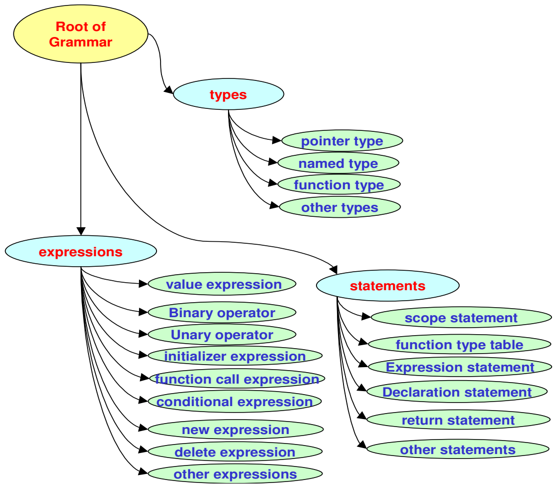

Figure 1: A simplified representation of the class hierarchy of classes representing the C++ grammar.

3.1 Building Grammars

For our purposes, a specification of a grammar is a set of product rules expressed in terms of terminals and non-terminals to define a language's constituent elements. BNF notation is a common form for the expression of such rules. **ROSETTA** represents a class library of terminals and nonterminals used to define a grammar. To each grammatical element (terminal or nonterminal object) in the **ROSETTA** application we associate source code. When the *Meta-level* application using the **ROSETTA** library is executed it produces source code which can be used to build an AST. ROSETTA's automatically generated parsers permit the creation of higher-level ASTs automatically from the lower level C++ grammar's AST (parsing from EDG's AST is provided as part of ROSE and Sage II). The hierarchy of classes represented by this source code is what we consider to be the *implementation* of the grammar. The default behavior is to build the SAGE II library (in a modified form) representing an implementation of classes defining the C++ grammar.

3.1.1 Building the C++ Grammar

It is relatively trivial (but lengthy) to define the C++ grammar in terms of terminals and nonterminals and associate with the terminals and nonterminals source code that implements those objects. The default grammar is the C++ grammar and the source code associated with it is essentially a modified form of the SAGE II source code (though automatically generated). We consider an implementation of the grammar to be a library of classes representing the different language elements defined by a grammar (all possible statements, expressions, types, symbols, etc.). We use a modified form of the Sage II library as the implementation of the C++ grammar, but other libraries that implement grammars and form the basis of different sorts of compiler tools exist[6, 5].

Figure 1 shows a simplified representation of the class hierarchy associated with the C++ grammar as

```
// Examples of grammatical elements for "Array" Grammar
   Grammar Array("Array");

// Construction of Terminal objects for "Array" grammar
   Grammar::Terminal ArrayAssignOp ("ArrayAssignOp","Array");
   Grammar::Terminal ArrayAddOp ("ArrayAddOp","Array");
   Grammar::Terminal ArraySubtractOp ("ArraySubtractOp","Array");
   Grammar::Terminal ArrayMultiplyOp ("ArrayMultiplyOp","Array");
   Grammar::Terminal ArrayDivideOp ("ArrayDivideOp","Array");
   .
   .
   .

// Construction of NonTerminal objects for "Array" grammar
   Grammar::NonTerminal ArrayBinaryOp ("Array");
   ArrayBinaryOp = ArrayAssignOp   | ArrayAddOp      |
                   ArraySubtractOp | ArrayMultiplyOp |
                   ArrayDivideOp   | ArrayAndOp | ArrayOrOp;
```

Figure 3: Example Meta-Program specification of Terminal and NonTerminal objects for "Array" grammar. Alternatively, higher level mechanisms in **ROSETTA** can automatically generate equivalent code from a class definition for the "Array" object.

defined using **ROSETTA**. The actual hierarchy of classes within the C++ grammar would include several hundred or more additional classes to represent all the specific operators etc. (terminals and non-terminals within the definition of the grammar). It is not practical within these figures to represent the full complexity of the C++ grammar or the higher level grammars which we additionally build.

3.1.2 Building Higher Level Grammars

Figure 3 shows examples of the declaration of terminals and non-terminals associated with an example "Array" grammar. To simplify the figures we will associate the letter **X** with the array object and build an **X** grammar. Clearly **X** could stand for any library abstraction. Figure 2 shows the modification of the corresponding simplified C++ grammar to build a higher-level grammar specific to a user-defined abstraction, **X**, note that the grammar includes **X** types, **X** statements, and **X** expressions. An AST built with this grammar clearly identifies language elements based on the **X** abstraction. As in the C++ grammar previously, in the *actual* **X** grammar a few hundred additional terminals and non-terminals must be added to address the full complexity of the C++ language (the full hierarchy of the classes defining the grammars would make the figures overly complex). Within the AST defined by the higher level grammars, searching for **X** statements for an arbitrary user defined abstraction, **X**, is simple because of the natural classification that results from the re-organization of the C++ AST into an AST.

Since higher-level grammars use the same source code base for their generated code, the explicit re-specification is not required except to add additional terminals and non-terminals to define the higher level grammar. In our motivating array class example we define the array grammar with respect to the C++ grammar and using a system of constraints. For example, the array user-defined type is represented in the array grammar by a C++ grammar's **class type** combined with a constraint that the name of the user-defined type was "Array". Additionally, within the array grammar we add as new terminals and non-terminals the public member functions of the array objects so that they could be identified as formal elements of the array grammar within expressions and statements and be clearly represented within the AST associated with the array grammar. Such specification of additional terminals and non-terminals can be automated from the class definition (the header file) which is parsed and known at runtime of the C++ *Meta-program*. The process means that grammars can be automatically generated from class definitions. This greatly simplifies the construction of library specific grammars.

Thus far we have shown how to build an **X** grammar for the array object, but a separate one should be considered to be built for the `Range` object so that it too, as an the array class abstraction, can be recognized at compile-time.

3.2 Connections between Grammars

Figure 4 shows how the individual grammars are connected in a sequence of steps; automatically generated parsers parse lower level grammars into higher level grammars. The initial AST using the C++ grammar is built by the EDG front-end from a C++ application code. The following describes the steps:

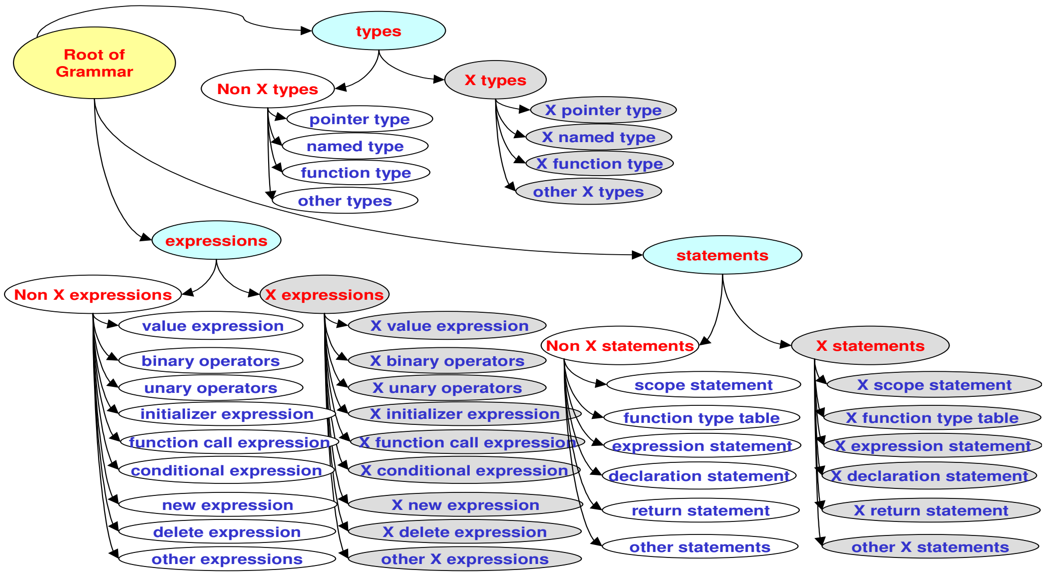

Figure 2: A simplified representation of the class hierarchy of classes representing the higher-level grammar associated with a user-defined X abstraction.

1. The first step generates the EDG AST, this program tree has a proprietary interface and is parsed in the second step to form the C++ Grammar's AST.

2. The C++ Grammar is generated by ROSETTA and is essentially comformant with the SAGE II implementation. This second step is representative of what SAGE II provides and presents the AST in a form where it can be modified with a non- proprietary public interface. At this second step the original EDG AST is deleted and afterwards is unavailable.

3. The third step is the most interesting since at this step the C++ Grammar's AST is parsed into higher level grammars. Each parent grammar (lower level grammar) parses itself into all of its child grammars so that the hierarchy of grammars is represented by corresponding ASTs (one for each grammar). Transformations can be applied at any stage of this third step and modify the parent AST recursively until the AST associated with the original C++ grammar is modified. At this point, an AST has been built using the `Array` and `Range` grammars (X Grammars), which is specific to the `Array` and `Range` objects contained within the A++/P++ array class library. The X Grammar AST not only identifies all `Array` and `Range` objects, but more importantly identifies all `Array` and `Range` expressions and `Array` and `Range` statements. For statement by statement optimizations `Array` and `Range` statements can now be easily recognized by traversing the AST. At the end of this third step all transformations associated with `Array` statements have been applied.

4. The fourth step is simply to unparse the AST associated with the C++ AST to generate optimized C++ source code. This completes the source-to-source preprocessing.

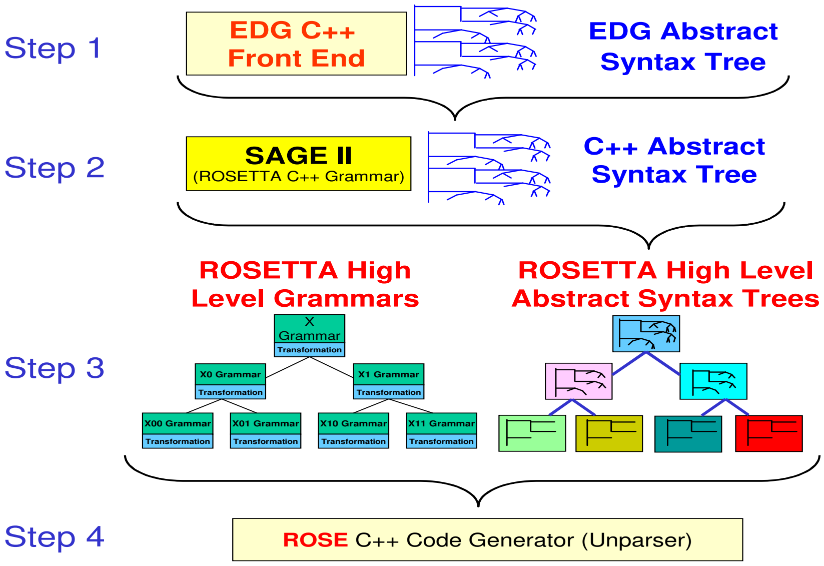

Figure 4: The connection of grammars (and parsers) representing the EDG front-end, SAGE and higher-level abstraction specific grammars built by ROSETTA.

An obvious next and final final step is to compile the resulting optimized C++ source code using the vendor's C++ compiler.

3.3 Connection to ROSE

ROSE provides for the specification of transformations and the automated introduction of such transformations into application source code. More information specific to **ROSE** can be found in [1]. The coupling of **ROSETTA** with **ROSE** provides the more complete source-to-source optimization mechanism with which to introduce library/framework or architecture dependent optimizations.

3.4 The *Meta-program* Level

A *Meta-program* level is used to build the source code that will be compiled to be the preprocessor; the *Meta-program* is a simple C++ program. The *Meta-program* specifically defines the construction and manipulation of grammars using the **ROSETTA** library and the Backus Naur Form (BNF) like abstractions within **ROSETTA**. The output of the *Meta-program*, when it is executed, is itself source code (written to two files). This resulting source code is compiled, with the ROSE infrastructure, to form a preprocessor specific to a given framework. The *Meta-program* can automatically generate a lot of source code, typically 200,000 lines, but it can be compiled in under a minute and once built into a preprocessor, by the library developer, need not be recompiled by application developers.

4 Implementation

The implementation of ROSETTA builds upon SAGE II [8], which is built upon the Edison Design Group (EDG) C++ front-end. Our work has been greatly simplified by access to these two tools. ROSETTA uses a modified form of the SAGE II which we have developed. The purpose was to

- Permit the automate generation of what is essentially a modified version of SAGE II
- Maintain the SAGE II source code (so that we can fix minor bugs and make additions (templates, and support for new C++ features as supported by EDG))
- Introduce the use of STL (as an outside library) into the design of SAGE II
- Remove as many asymmetries from the implementation of SAGE II so that the generation of the code could be simplified.
- Modify the SAGE II source to permit it to be used as a basis for all higher level grammars. This required naming the classes so that multiple grammars could coexist (to build hierarchies of grammars) in the same source-to-source compiler.

While using SAGE II as a basis for the grammars that ROSETTA generates, ROSETTA adds the significant capability to define grammars at the level of BNF notation. C++ classes are used to represent terminals and non-terminals and whole grammars.

5 Results

We have built high level grammars and used them to recognize and classify the use of user defined abstractions with numerous applications. The approach is particularly simple since the grammars can be built automatically from the library header files where the abstractions (C++ classes) are defined. Some additional information is required if numerous default definitions are to be overridden. It is not possible within this paper to present the ASTs for the higher level grammars since graphs as complex as these are difficult to visualize and we currently lack mechanisms for their presentation except for debugging purposes. At present we have processed approximately 1.5 Million lines of code through the tools built by **ROSETTA**. Current work has been to expand the complexity and

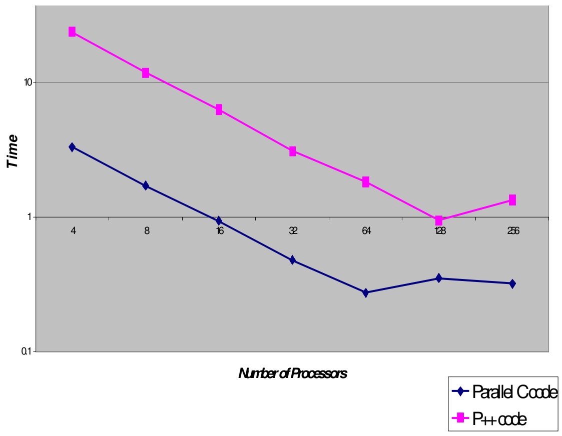

Figure 5: Parallel performance of stencil array statement with and without transformation. More sophisticated cache-based transformations are also possible.

quantity of source code being used as tests within this research work.

The most important use of this work has been in combination with other mechanisms within ROSE. Using grammars built by **ROSETTA**, and in conjunction with ROSE, full optimizing preprocessors have been built to optimize the performance of the A++/P++ array class library. Significant speedups were obtained depending on the array sizes; final performance matched that of C and FORTRAN performance.

Figure 5 shows the range of performance associated with different size arrays for the simple five point stencil operator (our motivating example) on the IBM Blue Pacific Computer. The results are in no way specific to this array statement, moderate and large size applications have been processed using preprocessors built with ROSE. The results show the time (vertical axis) plotted vs. number of processors for the motivating example problem using the P++ array class library (P++ code). The lower plotted line (Parallel Code) shows the transformed code, demonstrating the improved performance associated with the representation of the P++ code.

6 Related Work

A distinguishing feature of our work is that we automatically generate domain-specific grammars for object-oriented frameworks or applications. Such grammars include abstractions from object-oriented frameworks which are not a part of the C++ grammar. These grammars are built on top of the C++ grammar, using similar modified SAGE II source code as for the C++ grammar. In contrast, other work defines a single grammar representing the grammar of the base language itself (nothing higher level or user-defined abstraction specific) MPC++[6], NESTOR[5],SAGE[8]. As a result ROSETTA not only builds the source code restructuring tools specific to the C++ language (the base language) but also source code restructuring tools specific to the targeted domain-specific library/framework. This essentially provides a customized library/framework specific source code restructuring tool for the library/framework.

7 Conclusions

The use of object-oriented frameworks can often require or benefit from compile-time optimization if the abstractions are not sufficiently coarse grain and the context of the abstraction's use is important to the optimization. Examples include array class libraries (A++/P++, POOMA, Blitz, etc.), matrix class libraries (MTL, TNT, etc.), and complex grid geometry oriented frameworks like Overture[2]. This is the case for numerous sorts of abstractions for which the statements that use them consist of multiple expressions. Alternatively, blocks of statements may benefit from optimizations where their context relative to one another can only be seen at compile time. Our approach is particularly effective for array class libraries or higher level curvilinear grid libraries that include more sophisticated mathematical operators (e.g. div, grad, curl, laplacian, etc.). Examples could be array class, matrix classes, particle classes, finite-element classes, etc.

One of the limitations of this approach is that the construction of grammars through the constraining of the base level language grammar (the C++ grammar) does not permit the addition of new keywords to the C++ language. **But this is precisely a strength of our approach.** We don't want to add new features to the base language or provide a mechanism to simplify this. To do so would be to open the compiler in a fashion that would permit applications to be built that would rely upon specific language extensions, this would be counter productive to the development of portable standardized object-oriented libraries. Our goal is restricted to the optimization of existing object-oriented libraries/frameworks. Providing such a more sophisticated mechanism to extend C++ would simplify the addition of new keywords and language features but would be inconsistent with the use of the existing EDG front-end and parser from EDG to SAGE II. Such work would increase the complexity of ROSETTA well beyond practical limits.

Since a library can not readily see the context of how its elements are juxtaposed, only a compile-time tool can be expected to discern the use of object-oriented abstractions relative to one another within a user's application. With the abstract syntax tree (AST) exposed, clearly a relatively simple pattern matching approach could be used to identify the objects within an applications, but this is not enough to be useful. To recognize where transformations can be automatically introduced it is required that the use of the object-oriented abstractions be identified and classified into specific language/grammatical elements (expressions, statements, types, symbols, etc.). With this level of detail the AST is greatly simplified and can be traversed with the intent of abstraction dependent optimization, syntax checking, etc.

8 Special Thanks

We would like to thank the developers of the EDG front-end and SAGE II upon whose work we have based our own work for the last several years. Despite significant work to extend Sage II for our own purposes, it has been a significant asset to us.

References

[1] D. Quinlan, *ROSE: Compiler Support for Object-Oriented Frameworks*, Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000. Also published in special issue of Parallel Processing Letters (available soon).

[2] Brown, D., Henshaw, W., Quinlan, D. *"OVERTURE: A Framework for complex geometries"* Proceedings of the ISCOPE'99 Conference, San Fransisco, CA, Dec 7-10 1999

[3] J.Siek, A. Lumsdaine *"The Matrix Template Library: Generic Programming Approach to High Performance Numerical Linear Algebra"* Proceedings of the ISCOPE'98 Conference, Santa Fe, NW, Dec 8-11 1999

[4] S. Muchnick, *"Advanced Compiler Design and Implementation"* Morgan Kaufmann Publishers, July 1997

[5] Georges-Andre Silber, `http://www.ens-lyon.fr/~gsilber/nestor.`

[6] Ishkawa et. al. *Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach -*. In *Proceeding of Reflection'96 Conference*, April 1996 more info available at: http://pdswww.rwcp.or.jp/mpc++/mpc++.html

[7] Shigeru Chiba *Macro Processing in Object-Oriented Languages* In Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98), Australia, November, IEEE Press, 1998. more info available at: http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html

[8] F. Bodin et. al. *Sage++: An object-oriented toolkit and class library for building fortran and c++ restructuring tools.* In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994.

[9] Edison Design Group http://www.edg.com

[10] Todd Veldhuizen *Arrays in Blitz++* In *Proceeding of the Second International Symposium, ISCOPE 98*, Santa Fe, NM December 1998

[11] Karmesin, et al. *Array Design and Expression Evaluation in POOMA II.* In *Proceeding of the Second International Symposium, ISCOPE 98*, Santa Fe, NM December 1998

[12] Bassetti, F., Davis, K., Quinlan, D. *Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures* In *Proceedings of the ISCOPE'98 Conference*, Santa Fe, New Mexico, Dec 13-16 1998

[13] Lemke, M., Quinlan, D., *P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications* In *Proceedings of the CONPAR/VAPP V*, September 1992, Lyon, France; published in *Lecture Notes in Computer Science*, Springer Verlag, September 1992.

[14] Bassetti, F., Davis, K., Quinlan, D. *Toward FORTRAN 77 Performance From Object-Oriented C++ Scientific Frameworks* In *Proceedings of the HPC'98 Conference*, Boston, Mass. April 5-9, 1998

[15] Bassetti, F., Davis, K., Quinlan, D. *A Comparison of Performance-enhancing Strategies for Parallel Numerical Object-Oriented Frameworks* In *Proceedings of the first International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Marina del Rey, California, Dec, 1997