ROSE: The Design of a General Tool for the Independent Optimization of Object-Oriented Frameworks

K. Davis, B. Philip, D. Quinlan

This article was submitted to
International Symposium on Computing in Object-Oriented Parallel Environments, San Francisco, CA, December 7-10, 1999

May 18, 1999

ROSE: The Design of a General Tool for the Independent Optimization of Object-Oriented Frameworks

Kei Davis Bobby Philip Dan Quinlan

May 18, 1999

Abstract

ROSE represents a programmable preprocessor for the highly aggressive optimization of C++ object-oriented frameworks. A fundamental feature of ROSE is that it preserves the semantics, the implicit meaning, of the object-oriented framework's abstractions throughout the optimization process, permitting the framework's abstractions to be recognized and optimizations to capitalize upon the added value of the framework's true meaning. In contrast, a C++ compiler only sees the semantics of the C++ language and thus is severely limited in what optimizations it can introduce. The use of the semantics of the framework's abstractions avoids program analysis that would be incapable of recapturing the framework's full semantics from those of the C++ language implementation of the application or framework. Just as no level of program analysis within the C++ compiler would not be expected to recognize the use of adaptive mesh refinement and introduce optimizations based upon such information.

Since ROSE is programmable, additional specialized program analysis is possible which then compliments the semantics of the framework's abstractions. Enabling an optimization mechanism to use the high level semantics of the framework's abstractions together with a programmable level of program analysis (e.g. dependence analysis), at the level of the framework's abstractions, allows for the design of high performance object-oriented frameworks with uniquely tailored sophisticated optimizations far beyond the limits of contemporary serial FORTRAN 77, C or C++ language compiler technology. In short, faster, more highly aggressive optimizations are possible. The resulting optimizations are literally driven by the framework's definition of its abstractions. Since the abstractions within a framework are of third party design the optimizations are similarly of third party design, specifically independent of the compiler and the applications that use the framework.

The interface to ROSE is particularly simple and takes advantage of standard compiler technology. ROSE acts like a preprocessor, since it must parse standard C++[1], and its use is optional, it can not be used to introduce any new language features. ROSE reads standard C++ source code and outputs standard C++ code. Its use is always optional, by design, so as not to interfere with and to remain consistent with the object-oriented framework. It is a mechanism to introduce optimizations only; **adding language features using ROSE is by design no more possible than within the framework itself.** Importantly, since ROSE generates C++ code it does not preclude the use of other tools or mechanisms that would work with an application source code (including template mechanisms).

1 Introduction

The development of object-oriented frameworks has permitted the centralization of expertise and its reuse by numerous people, research groups, institutions and industries. The expertise represented within object-oriented frameworks has ranged from support for complex geometries and grid generation to the introduction of advanced numerical algorithms (such as adaptive mesh refinement), to the encapsulation of parallelism on advanced computer architectures. The reuse of this work within Overture framework has extended from Computational Biology at UC Davis, to the modeling and design of sails for America's Cup Yacht Racing at Doyle (sp?), to the design of diesel engine simulations at Lawrence Livermore National Laboratory with Caterpillar Inc. Other frameworks have likely also received broad use spanning multiple research and industrial disciplines and leveraging their support. The extream breadth of the different areas of expertise represented by individual object-oriented frameworks has not been entirely offset by the performance

[1]ISO/IEC 14882:1998 C++ standard as implemented by EDG

issues associated with high performance computing, particularly at national laboratories where high performance within computational simulations is in greater focus (and the parallel computer architectures more specialized, complex and obscure).

In looking at what has been learned from the use of object-oriented frameworks, it has been the extream elegance of the powerful abstractions (grids encapsulating complex moving geometry, solvers encapsulating adaptive mesh refinement, and array operations encapsulating parallelism, etc.) when layered with the syntactic sugar represented by overloaded operators that has enticed framework and application developers the most.

In contrast to the abstractions in an object-oriented framework, the C++ compiler ignores the rich semantics embedded in these powerful abstractions and effectively filters their potential influence upon the optimization of applications using such object-oriented frameworks. For example, within an array class the semantics of the array abstractions may include complete evaluation of the *rhs* before assignment to the *lhs*, array semantics; but the compiler can not see this and so cannot use this semantics to introduce optimizations. The array class might enforce array semantics, but the compiler would never be sure that loop optimizations could use such knowledge, the result is suboptimal performance. Within advanced cache optimizations the lack of such seemingly detailed information can invalidate there use, dramatically effecting the performance on cache based architectures.

ROSE represents a mechanism to capitalize on the semantics represented by abstractions implemented by an object-oriented framework. ROSE uses this semantic information to define an automated mechanism to introduction optimizations which FORTRAN and C compilers have insufficient semantic information to introduce. Access to this additional semantic information, beyond that of the C or C++ language semantics is the key reason why advanced transformations for cached based architectures can be automated. Such cache transformations can out perform typical FORTRAN 77 and C compilers by factors of three to four /cite[CacheOptimization]. Serial optimizations are only half of the optimization potential, parallel object-oriented frameworks define specific parallel semantics for their abstractions. Parallel optimizations are also possible (for example, scheduling of communication), but inaccessible since the base language's semantics (for example, C++) does not include parallel semantics. Serial compilers can implement optimizations that take advantage of parallel semantics that they know nothing about and can not see. ROSE provides a mechanism to capture the original high level semantics introduced by the object-oriented framework and use these to drive more sophisticated optimizations that those possible within the greatly restricted semantics of the base language (C++, for example).

1.1 Object-Oriented Frameworks

The development of scientific simulations generally involves: 1) the definition of the problem 2) the specification of the relevant physics 3) the selection of numerical approaches 4) the design of the application code 5) the specification of the geometry upon which to run the simulation. A measurable percentage of application are initially conceived of within restricted geometries (Cartesian coordinates or simple transformations of Cartesian coordinates), but many or most practical applications involve some geometric complexity. Despite the emphasis on high performance computing the development of the grid generation consumes many months compared the the execution time of the computations which take only days. The wide spread use of simulations on complex geometries is limited by the availability and complexity for users to master such techniques and the ability of current computational hardware to process such simulations.

Overture is an object-oriented simulation framework that addresses many aspects of the development of complex simulations associated with industrially relevant applications. A current focus within the Overture team is the simulation of diesel engines. Such engine simulations involve significant complexities, with complex internal shapes and many moving parts. The time scales and relative size of features within the geometry and the fluid flow require locally tailored grid resolution which demand adaptive mesh refinement numerical methods. The complexity of tying these complexities together can not overshadow the requirements of high performance on advanced computer architectures.

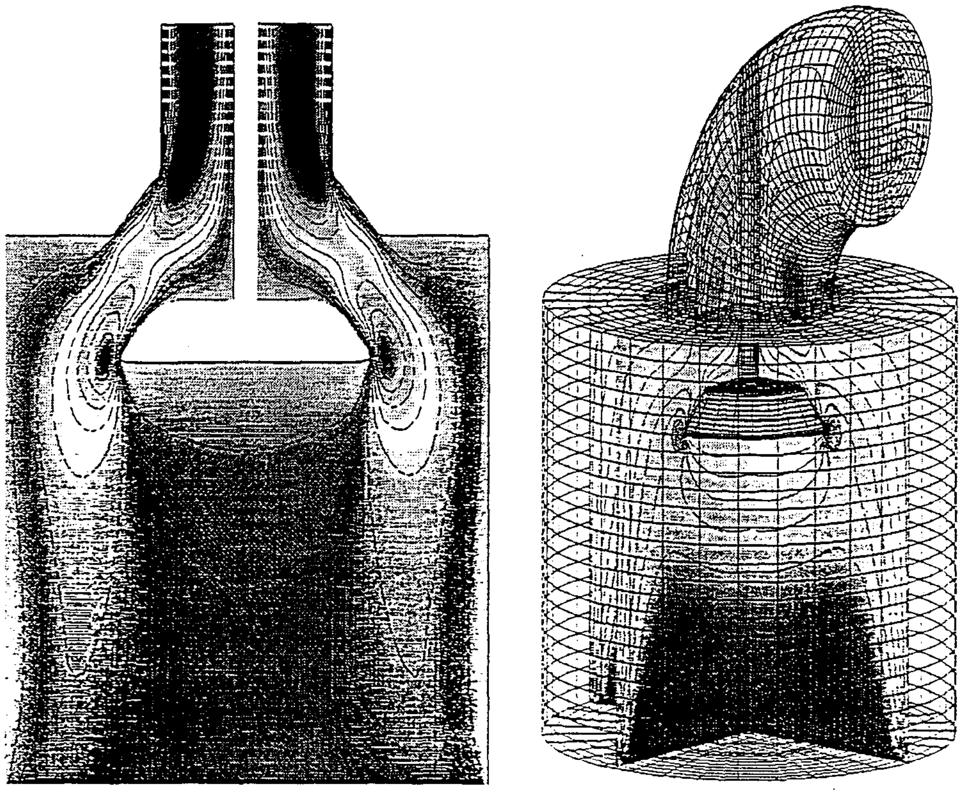

Figure 1: Example Overture application: flow simulation of moving valve within an engine in 2D and 3D. Such simulations tie together numerous software complexities. The expression of such complex flow applications using abstractions represented within the Overture framework provides a rich opportunity for sophisticated optimizations, but for the fact that the semantics of the framework's abstractions are ignored by the underlying C++ compiler. ROSE provides a mechanism to use the framework's semantics to automate sophisticated optimizations.

1.2 What Problem ROSE Solves

The purpose of ROSE is to provide a mechanism to achieve high performance for scientific computations. ROSE addresses the specific and ever changing requirements of current computer architectures by abstracting the details associated with performance (specific transformations) and separating them from the development of the more general application. The right transformation on the wrong architecture would of course only slow performance generally, so separating these as much as is reasonable provides a mechanisms to develop scientific software that are portable. Using object oriented frameworks the development of the scientific applications (such as those using Overture) is greatly simplified, ROSE addresses performance issues within the use of such object-oriented frameworks. ROSE is not in any way specific to use with Overture but is being use with Overture to address provide optimizations cache based optimizations, these optimizations are independent of Overture and can be applied to other frameworks and even C language applications directly /footnoteThe use of ROSE with C applications as a mechanism to automate the introduction of cache based transformations has not been the target of current work and would require either greater program analysis that is likely possible today or some direct intervention by the application writter to specify where ROSE could apply the optimizations (transformations). This work would expand the applicability of ROSE orthogonally to the optimization of object-oriented frameworks..

Specific problems that ROSE addresses can include any compile time analysis or transformation, a short list includes:

- Simple elegant interface, consistent across all machines

- Loop fusion of binary operators
- Loop fusion across statement (requires simple dependence analysis)
- Cache based transformations
- Temporal locality optimizations
- compile-time performance metrics

1.3 Scope

ROSE applies broadly to all C++ object oriented frameworks, nothing in the design makes is specific to A++/P++ or Overture directly. Since C is a subset of C++ it could be used more generally to introduce specialized optimizations into C applications as well. Avoiding the use of the object-oriented framework and the abstractions that it presents however complicates the specification of where optimizations can be preformed. So while ROSE could work with C as a mechanism to introduce transformations the C program application would have to specify in some way where optimizations would be done. The effect of avoiding the object-oriented frameworks is to lose the abstractions that can drive the automated optimizations. Applications of ROSE to other languages is clearly possible, but not within the scope of our current work.

ROSE provides optimizations through the introduction of transformations. The users that ROSE targets are those developing object-oriented frameworks, and not the users of those frameworks. Application developers would have no use for ROSE except to implement optimizations not implemented by the framework writter. Since ROSE is entirely optional, it does not change the semantics of the framework in any way, it only addresses performance optimizations.

1.4 Short Term Goals

The immediate goal of current work is to provide performance optimizations for Overture (and more specifically the A++/P++ array class library used within Overture). Specific transformations currently being worked on include simple binary operator elimination, simple spatial transformations for cache, loop fusion and more sophisticated temporal transformations for cache.

2 How ROSE works

ROSE acts like a preprocessor, it does not introduce any language features, it accepts C++ source code and outputs C++ code. Its use is by design optional so as not to interfere with with the object-oriented framework. It is a mechanism to introduce optimizations only, adding language features using ROSE is by design not possible. Internally within ROSE each transformation represents a single pass. Currently we assume that all transformations are independent (mechanisms for staging of transformations would be more complex and may be represented in later work). Within the presentation of how ROSE works we will consider only a single transformation, as an example we will consider the transformation of a simple 1D array statement (but most of our presentation will be more abstract).

ROSE internally uses the Sage II source code restructuring tool from University of Indiana and ISI (Gannon and Keselman). Sage uses the Edison Design Group (EDG) C++ front end, and provides a public interface to the internal (private) EDG representation. Essentially, Sage II implements the C++ grammar as an object-oriented interface (each nonterminal is an object), the user's C++ application is then internally represented as a program tree (graph) consisting connected elements of the C++ grammar.

The specification of a transformation (optimization) within an arbitrary object-oriented framework consists of two parts:

1. the specification of when it can be introduced, and
2. the specification of the transformation itself.

Many other transformations proceed nearly identically except for the specification of the transformation itself.

2.1 Specification of where to introduce optimizations

The specification of where to introduce a transformation typically requires the use of program analysis. Simply put, program analysis is the discovery of how the application code (using the semantics of the programming language) is using variables and language constructs (loops, conditionals, etc.). Much work has been done on program analysis and within compilers it is quite sophisticated, but surprisingly little is possible to fathom from an application using only the semantics of a low level language (C, C++, FORTRAN). In general, the most complete program analysis occurs where the languages semantics is most restrictive. Clearly the languages with the most restrictive semantics tend to complicate the expression of sophisticated scientific applications and this is the basis of FORTRAN often out performing other more modern languages.

The basis for knowing when to introduce a transformation is to perform some sort of pattern matching. Explicit pattern matching is not particularly practical nor sufficiently general, a more acceptable approach is to define a grammar. All languages are defined using a grammar, a standard way to present a grammar is to use Bachkus-Nour Form (BNF) notation[2]. An object-oriented framework also has a grammar and its grammar can be simple to complex depending upon the framework. But since the development of an object-oriented framework does not proceed along the same conventional lines as the development of a language, few if any object-oriented frameworks have an explicit grammar (we present a representation of the grammar for an array class separately).

Clearly C++ has a grammar, internally compilers use this grammar as part of the implementation of the compiler and the internal representation of the application source at internal stages within the compilation process (more than one grammar is often used). The access to the internal program representation is an important part of ROSE. ROSE uses the Sage II source code restructuring program to provide access to the program tree. Sage uses the EDG front end and separate agreements with EDG have permitted us access to all EDG source. Sage II provides a public interface to the private EDG internal representation, providing a object-oriented implementation of the C++ grammar used to define any C++ application. Sage permits the editing of that internal representation and the unparsing of its representation back into C++ source. The object-oriented implementation of the C++ grammar (each terminal and nonterminal is a separate object, etc.) provides the base level grammar within ROSE.

2.1.1 How The Grammar Is Defined

Since the C++ grammar is quite large (consisting of several hundred of elements) the C++ representation of the program tree can be overwhelmingly complex. The direct identification of abstractions (as implemented in C++) of framework abstractions is far too complex. To simplify the job a grammar is defined which represents the object-oriented framework directly. Using ROSE with an arbitrary object-oriented framework would require the specification of the grammar representing the targeted framework. This is done using a conventional Extended BNF notation (a classic way to define grammars).

In the example of a transformation of array assignment statements two grammars will be defined. The first is a grammar is defined to represent the array class. Space within this paper does not permit the presentation of this grammar (it is available one the Overture web site separately). Figure x shows a small part of the grammar representing the A++/P++ array class library (the Array Grammar), its complete listing is available separately through the Overture web pages. This figure shows the use of the grammar definition mechanism within ROSE which allows a grammar to be defined using Extended Bachus-Nour-Form (EBNF) notation. This mechanism for specifying a grammar is particularly simple because it permits conventional BNF notation to be used, the execution of this C++ code represent the grammar is sufficient to build the code that is then compiled with ROSE to define the use of this grammar.

Additional grammars that are specific to each individual transformations can now be defined which use the array grammar. Figure x shows the *array assignment grammar.*

More complex transformations (stencils, for example) require still higher level grammars that build on top of this. Such example would be too complex for presentation within this paper.

[2]For more background see the Dragon Book

```
Grammar::NonTerminal ArrayExpression =
    ArrayNumericExpression
    | ArrayRelationalExpression
    | ArrayLogicalExpression
    | C_Expression
    | ArrayOperand;
```

Figure 2: Example of product rule (in EBNF notation) for nonterminal of the *array grammar* using the mechanisms for defining grammars within ROSE.

```
Grammer::NonTerminal arrayOperator = arrayBinaryOperator;
Grammer::NonTerminal assignmentOperator = arraytype "::" "operator="
Grammer::NonTerminal transformableExpression;
transformable_expression = transformableExpression & operator & transformableExpression |
                           arrayOperand arrayOperator arrayOperand | arrayOperand;
Grammer::NonTerminal lhs_operand = arrayOperand;
Grammer::NonTerminal rhs_operand = transformable_expression;
Grammer::NonTerminal transform_statement = lhs_operand & assignmentOperator & rhs_operand;
```

Figure 3: Array Assignment Grammar Production Rules: Example of product rules (in EBNF notation) for the *array assignment grammar* using the mechanisms for defining grammars within ROSE.

2.1.2 How the grammar is used

Within ROSE the specification of the grammar is sufficient to generate code that ROSE then uses internally to build the parser and the object oriented implementation of the grammar used to represent a program tree using the associated grammar. Tens of thousands of lines of code can be automatically generated, greatly simplifying the process of handling large grammars associated with sophisticated object-oriented frameworks. part of ROSE is the user interface which permits customization of the behavior of these implementations of the grammar, permitting user defined code to be inserted into the implementations of the grammars. Current attempts within this research are to automate as much of the generation of the grammars as possible.

The introduction of grammars provides a mechanism to separate different parts of the application code. The array grammar is used to define array operations (statements, expressions, types, etc.). Sage II provides a program tree represented by a graph of elements in the C++ grammar. This representation is not sufficiently refined for our purposes and does not effectively differentiate parts of the program tree representing non-array class code from parts of the program tree representing array class code (though clearly it is all C++ code). To further refine and synthesize the application code's use of the array class the program tree is parsed using the array class grammar, effectively filtering out all non-array class code from the C++ application.

As a rule, if a program tree can be parsed (without error) using a grammar then the program tree has a representation in that grammar. More specific to the array class, if a part of an application code parses using the array class grammar then it represents application code that uses the array class. This simple mechanism is used to literally find the use of array class abstractions (array objects, operators, etc.) within the application source. Once found we know only that array class abstractions have been located, nothing more specific or specific enough to pinpoint where to introduce a transformation. Parsing the C++ grammar into a representation of the application using the array class grammar builds a new program tree.

The specification of the additional grammar is required to pinpoint array assignment statements (for example) within the array class grammar representation of the program tree. The process of defining the array-assignment grammar and parsing the array grammar program tree into an array-assignment grammar program tree proceeds i-dentally as in the transition from the original C++ grammar (the Sage II program tree). The program tree using the array grammar is parsed and a program tree using the array assignment grammar is constructed.

With the application program's representation in the array-assignment grammar it is clear that all array assignment statements have been found, since by definition they were recognized by the array assignment grammar. The use of still higher level grammars can be used to further refine (filter) the collection of array assignment statements. Higher level grammars might include stencil grammars for example which would pinpoint specific types of stencil operations for cache based transformations.

2.2 Specification of a Transformation

The specification of a transformation addresses what the optimization will be and completes the process of defining an optimization once it is known where it will be introduced. The specification of the transformation must be represented in multiple parts and these parts must be assembled depending upon the the context of the original statement (in the case of an array assignment optimization the context can include the dimension, number of operands, etc.; if this information is not represented in the grammar directly).

The specification of the transformation varies greatly in complexity, it is the program tree that is transformed and the tree consists of many different elements of grammar, each is transformed separately. The transformation of the program tree occurs as a transformation of the program tree associated with a higher level grammar into a program tree associated with a lower level grammar.

For the case of a transformation of an array-assignment statement the transformation consists of a transformation of the program tree associated with the array-assignment grammar into a program tree associated with the array grammar and then a second transformation of the program tree associated with the array grammar.

For each element of grammar within the program tree a transform function is defined (current attempts are being made to automate the generation of these functions from the definition of the grammar (directly from the specification of the transformation rules). This transformation is defined to be a map of the elements of the higher level grammar into the lower level grammar.

There are two ways to specify the transformation of a terminal or non-terminal in a higher level grammar into a lower level grammar.

1. Hard coded Here the transformation is explicitly defined by supplementing the definition of that element of the grammar. Space precludes an example, but required elemets are assembled explicitly from the C++ grammar defined by the Sage II objects.

2. By example Here the transformation is separated into pieces and how the pieces are fix together is defined explicitly.

More mechanisms may be defined as research is done to more completely automate the specification of a transformation from as simple a description as possible.

In the case of an assignment statement the transformation of the different elements of the grammar are called recursively through the program tree to build a separate program tree representation within a lower level grammar.

A grammar is defined by a collection of product rules, but the transformation of one grammar into another is outside of what a common grammar can represent. So the transformation of a grammar into a lower level grammar is defined by a separate set of transformation rules. These rules are dependent only upon a given pair or grammars. Current research in ROSE is defining mechanism to automate the generation of code which implements these rules.

Figure x shows the specification of the elements of the transformation, Figure y shows how these are assembles at compile time.

Figure x shows how these elements of the transformation are assembled into code based on the details of the statement being transformed. This code defined the transform membe function on for the **arrayAssignmentStatement** element of the grammar. Many detials are left to the reader since this paper is of limited length.

As a final example, we present two tiny codes, one using the array class directly, and the other being the output of ROSE.

2.3 Summary

The final result in ROSE is a mechanism that reads C++ application source code, recognizes arbitrary abstractions and implements customized transformations that provide better performance. The mechanism

```
// Example of transformation specification (in this case for an array statement)

       FUNCTION_DEFINITION UNIQUE_PART_OF_TRANSFORMATION ()
          {
         // This code is only required once for all the operands in the same scope
            int GLOBAL_INDEX_NAME = 0;
          }

       FUNCTION_DEFINITION LHS_PART_OF_TRANSFORMATION ()
          {
         // This is code that is required once for the Lhs operand
            double* RESTRICT LHS_ARRAY_DATA_POINTER = LHS_ARRAY.getDataPointer();
          }

       FUNCTION_DEFINITION RHS_PART_OF_TRANSFORMATION (int numberOfRhsOperands)
          {
            LOOP(numberOfRhsOperands)
               {
              // This is code that is required once for the Lhs operand
                 double* RESTRICT RHS_ARRAY_DATA_POINTER = RHS_ARRAY.getDataPointer();
               }
          }

       FUNCTION_DEFINITION LOOP_PART_OF_TRANSFORMATION ()
          {
         // This code is required only once for this transformation
            const int base_1D_0   = LHS_ARRAY.getBase  (0);
            const int bound_1D_0  = LHS_ARRAY.getBound (0);
            const int stride_1D_0 = LHS_ARRAY.getStride(0);
            for (GLOBAL_INDEX_NAME = base_1D_0; GLOBAL_INDEX_NAME <= bound_1D_0; GLOBAL_INDEX_NAME++)
               {
              // This code would be modified with the edited user inner loop statement
              // The first statement will be replaced within the transformation process

              // LHS_ARRAY_DATA_POINTER[ROSE_SUBSCRIPT_COMPUTATION_1D(GLOBAL_INDEX_NAME,0,stride_1D_0)] =
              // RHS_ARRAY_DATA_POINTER[0];

              // It is in the transformation of the statement (called by ROSE recursively on
              // the program tree associated with the highest level grammar) that specific
              // transformations are introduced).  Look at the specification of the transformations
              // of the grammar elements (at all levels of the grammar higherarchy).
                 TRANSFORMED_STATEMENT();
               }
          }
```

Figure 4: Array Assignment Grammar Production Rules: Example of product rules (in EBNF notation) for the *array assignment grammar* using the mechanisms for defining grammars within ROSE.

is powerful since it is fully programmable, can be coupled with dependence analysis and any other traditional program analysis mechanism to capitalize upon the surrounding context of the use of a framework's abstraction. In this way, it significantly more powerful than C++ template based mechanisms (expression templates) which preform little more than text editing with no program analysis being possible, such methods delegate all program analysis to runtime checks.

Within ROSE the optimization of abstractions within an object-oriented framework can be defined through the specification of a grammar and a transformation. Some optimizations may be best introduced using multiple grammars defined within a hierarchy. The example presented in this paper shows how an array grammar is specified and how a grammar specific to the optimization of array assignment statements (the array assignment grammar) is defined using that array grammar. Our present experience has been with such shallow grammar hierarchies, but it is clear that the mechanisms being developed are general and can support the deeper hierarchies required for the specification of much more complex stencil specific optimizations.

The specification of an optimization requires the definition of one or more grammars and the specification of the transformation itself. The advantage of using a hierarchy of grammars is that higher level grammars are greatly reduced in size (number of terminals and nonterminals) and corresponding complexity. In addition multiple grammars defined above a common lower level grammar within the hierarchy can leverage the defined transformations of the lower level grammar, simplifying the development of specialized optimizations. This construction of a hierarchy of grammars supports the iterative refinement of increasingly

```
// Example of a transform member function
// Code to put the program tree fragments (after they are recognized
// using our transformation specification grammar)

TransformationSpecificationStatementBlock transformSpecification;
TransformResult arrayAssignmentStatement::transform()
  {
 // Build storage for final transformation
    TransformResult result;

    TransformationSpecificationFunction transformMetaFunctionUnique = transformSpecification.get("UNIQUE_PART_OF_TRANSFORMATION");
    transformMetaFunction = transformMetaFunctionUnique.edit ("GLOBAL_INDEX_NAME","i");
    result.add(transformMetaFunctionUnique.getBlock());

    TransformationSpecificationFunction transformMetaFunctionLhs = transformSpecification.get("LHS_PART_OF_TRANSFORMATION");
    transformMetaFunction = transformMetaFunctionLhs.edit ("LHS_ARRAY",variableName[0]);
    transformMetaFunction = transformMetaFunctionLhs.edit ("LHS_ARRAY_DATA_POINTER",transformedVariableName[0]);
    result.add(transformMetaFunctionLhs.getBlock());

    for (int i=0; i < numberOfRhsOperands; i++)
       {
         TransformationSpecificationFunction transformMetaFunctionRhs = transformSpecification.get("RHS_PART_OF_TRANSFORMATION");
         transformMetaFunction = transformMetaFunctionRhs.edit ("RHS_ARRAY",variableName[i+1]);
         transformMetaFunction = transformMetaFunctionRhs.edit ("RHS_ARRAY_DATA_POINTER",transformedVariableName[i+1]);
         result.add(transformMetaFunctionRhs.getBlock());
       }

    TransformationSpecificationFunction transformMetaFunctionLoop = transformSpecification.get("LOOP_PART_OF_TRANSFORMATION");
    transformMetaFunction = transformMetaFunctionUnique.edit ("GLOBAL_INDEX_NAME","i");
    transformMetaFunction = transformMetaFunctionLhs.edit ("LHS_ARRAY",variableName[0]);
    result.add(transformMetaFunctionLoop.getBlock());

    return result;
  }
```

Figure 5: Array Assignment Grammar Production Rules: Example of product rules (in EBNF notation) for the *array assignment grammar* using the mechanisms for defining grammars within ROSE.

specialized optimizations over time. An over-arching goal within ROSE has been to define a simple coherent mechanism that permits the implementation of optimizations within arbitrary object-oriented frameworks and to accomplish these optimizations in a few hours of work.

3 Conclusion

The mechanism presented within ROSE defines a general mechanism applicable to any object-oriented framework. The framework's development of powerful abstractions is not compromised by the inability of the compiler to optimize them. It is likely still that case that using abstractions of appropriate granularity is prudent in the design of object-oriented frameworks, but the interaction of abstractions of a framework can be optimized without resorting to obscure C++ template tricks and and the mechanisms are fundamentally more powerful. ROSE as a mechanism is more powerful because it permits the use of a full range of standard program analysis techniques and because the transformations can capitalize upon the surrounding context. This later point of understanding the surrounding context as part of the optimization is crucial to the development of fusion over statements and sophisticated cache optimizations addressing spatial and temporal locality within scientific computations.

The mechanisms within ROSE, since they are orthogonal and independent from the framework, can be retrofitted into existing frameworks for use with previously developed applications using those frameworks.

Lastly, ROSE preserves the intended elegance of the frameworks design by providing mechanisms for the optimization of the interaction of any size abstraction expressed using the C++ languages overloaded operators. Using the semantics of the framework's abstractions more sophisticated (and higher performance) optimizations are possible since knowledge of the framework's semantics leads directly to significantly more information and from that a greater level of optimization can follow.

```
#include "A++.h"

int main()
   {
     int size = 10;
     double gamma = 2.0;
     doubleArray A(size);
     doubleArray B(size);
     Range I(1, size-2);
     Range J(1, size-2);

     A(I) = ( B(I+1) + B(I-1) ) * 2.0;

     printf ("Program Terminated Normally! \n");
     return 0;
   }
```

Figure 6: Example A++ code before processing using ROSE.

4 Bibliography

[BCHQ] Brown, D., Chesshire, G., Henshaw, W., and Quinlan, D., "OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments," Published in Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[BQ] Balsara, D., Quinlan, D., "Parallel Object-Oriented Adaptive Mesh Refinement," Published in Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[PQ] Parsons, R., Quinlan, D., "A++/P++ Array Classes for Architecture Independent Finite Difference Computations," Proceedings of the Second Annual Object-Oriented Numerics Conference, Sunriver, Oregon, April 1994.

[BLQ] Balsara, D., Lemke, M., Quinlan, D., "AMR++, a C++ Object Oriented Class Library for Parallel Adaptive Refinement Fluid Dynamics Applications, Proceedings of American Society of Mechanical Engineers, Winter Annual Meeting, Anaheim, CA, November 8-13, Adaptive, Multilevel and Hierarchical Computational Strategies, AMD-Vol. 157, pg. 413-433, 1992

[LQ2] Lemke, M., Quinlan, D., "P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications", CONPAR/VAPP V, September 1992, Lyon, France, Lecture Notes in Computer Science, Springer-Verlag, 1992.

```
#include <A++.h>

#4 "test2.C"
int main()
   {
     auto int size=10;
     auto double gamma=2;
     auto doubleArray A(size);
#9 "test2.C"
     auto doubleArray B(size);
#10 "test2.C"
     auto Range I(1,size - 2);
#11 "test2.C"
     auto Range J(1,size - 2);

#13 "test2.C"
        {
       // Transformation for: A(I) = B(I-1) + B(I+1);
          int rose_index=0;
          double * restrict A_rose_pointer = (A . getDataPointer)();
          double * restrict B_rose_pointer = (B . getDataPointer)();
          const int base_1D_0      = (I . getBase)();
          const int bound_1D_0     = (I . getBound)();
          const int rose_stride    = (A . getStride)(0);
          const int rose_base      = (B . getBase)(0);
          for(rose_index = base_1D_0; rose_index <= bound_1D_0; rose_index++)
             {
               A_rose_pointer[rose_index] =
                    (B_rose_pointer[(rose_index + 1)] + B_rose_pointer[(rose_index - 1)]) * 2;
             }
        }

#249 "/usr/include/stdio.h"
     printf(((const char * )"Program Terminated Normally! \n"));
#49 "test2.C"
     return 0;
   }
```

Figure 7: Example of output from processing of A++ code using ROSE.