ROSE: Compiler Support for Object-Oriented Frameworks

Dan Quinlan[1]

Lawrence Livermore National Laboratory, Livermore, CA, USA,
dquinlan@llnl.gov,
WWW home page: http://www.llnl.gov/CASC/people/quinlan/

Abstract. ROSE is a preprocessor generation tool for the support of compile time performance optimizations of general object-oriented frameworks. Within this work ROSE is being applied first to **Overture**, a serial/parallel object-oriented framework for solving partial differential equations in two and three space dimensions. The optimization of the interactions between objects within **Overture** is of particular interest since the **Overture** applications can be computationally large (many millions of mesh points and iterating over thousands of time-steps). Unfortunately, optimizations that might be obvious to the framework developer or application developer (e.g. cache based optimizations), due to the precise semantics of the framework's abstractions, are often lost through the C++ compiler's inability to recognize or leverage such semantics. Preprocessing steps can be used to introduce transformations using the semantics of a framework's abstractions, but the development of such a preprocessor tool is particularly complicated for a general object-oriented language such as C++. This paper shows how such framework specific preprocessors can be automatically generated.
In this paper we briefly present **Overture** with some examples, and present our approach toward optimizing the performance for **Overture** and the **A++P++** array class abstractions upon which **Overture** depends. The results we present show that the semantics of the abstractions represented within **Overture** and the A++P++ array class library can be used to generate a preprocessor using ROSE. The results demonstrate the performance of an **Overture** application with and without such a preprocessing step, the final performance with preprocessing is equivalent to that of optimized C and Fortran 77. By design, ROSE is general in its application to any object-oriented framework or application and is in no way specific to **Overture** .

1 Introduction

ROSE is a programmable source-to-source transformation tool for the optimization of C++ object-oriented frameworks. In our work we target the **Overture** Framework specifically (www.llnl.gov/casc/Overture), a parallel object-oriented C++ framework for solving partial differential equations associated with computational fluid dynamics applications within complex moving geometries. Work

on the **Overture** framework represents our research in the modeling of diesel engine combustion. While we have specific goals for this work within **Overture**, ROSE applies equally well to any other object-oriented framework.

A common problem within object-oriented C++ scientific computing is that the high level semantics of abstractions introduced (e.g. parallel array objects) are ignored by the C++ compiler. Classes and overloaded operators are seen as unoptimizable structures and function calls. Such abstractions can provide for particularly simple development of large scale parallel scientific software, but the lack of optimization greatly effects performance and utility. Because C++ lacks a mechanism to interact with the compiler, elaborate mechanisms are often implemented within such parallel frameworks to introduce complex template-based and/or runtime optimizations (such as runtime dependence analysis, deferred evaluation, runtime code generation, etc.). These approaches are however not satisfactory since they either require long compile times (hours) or are not sufficiently robust.

ROSE represents a mechanism to build preprocessors that reads the user's application source code and outputs highly optimized C++ code. The output from a preprocessor built from ROSE is itself C++ code (but transformed using the semantics of the object-oriented abstractions represented within the framework). ROSE helps the framework developer define framework specific (or application specific) grammars (more specifically a hierarchy of grammars), one or more transformations can be associated with each grammar. The transformations assume complete knowledge of the serial and parallel semantics of the object-oriented abstractions and are thus safe by definition. Multiple program trees are then built, one for each grammar. The traversal of the much simpler program trees represented by the higher level grammars (as opposed to that of the C++ program tree) permits the identification of locations where transformations are then applied to introduce specific optimizations. The final modified program tree is then unparsed to generate C++ source code. The source code transformations can readily exploit knowledge of the architecture, parallel communication characteristics, and cache architecture in the specification of the transformations. Within **Overture** , a parallel framework, the serial and parallel semantics are known and transformations can range from serial loop optimizations to parallel message passing optimizations, threading could alternatively be automated with such transformations (where identified using the framework's parallel semantics).

We have developed this work as an optional alternative (optional since the framework's semantics are in no way modified through the use of the preprocessors built with ROSE) to the definition of standardized parallel languages. A language is harder to develop, more difficult to optimize, and difficult to get accepted into scientific computing.

ROSE is implemented using several other tools. We use the EDG C++ front-end and the Sage II source code restructuring tool. ROSE exists as a layer on top of Sage II (which represents an open interface to the C++ program tree provided though the EDG front-end), while Sage II exists as a layer on top of the EDG front-end. The EDG front-end is a commercial C++ front-end,

providing us with an implementation of the full C++ language (as complete as is available today). By design, we leverage many low level optimizations provided within modern compilers while focusing on higher level optimizations largely out of reach because traditional approaches can not leverage the semantics of high level abstractions. In doing so, we slightly blur the distinction between a library/framework and a language. Because we leverage several good quality tools the implementation is greatly simplified.

This paper presents our work to automate the construction of preprocessors specific to an arbitrary object-oriented framework, in this case the **Overture** framework. With such a tool the development of preprocessing mechanisms to leverage the semantics of a framework's abstractions can be handled as part of the framework development (by the framework developer or even users). Both significant optimizations (machine dependent cache based transformations) and more readily optimizible abstractions can be defined. The effect is to provide a readily customizable compilation mechanism that can leverage existing high-level user defined semantics (significantly beyond that of the C++ language's relatively general and low level semantics).

2 The Overture Framework

The **Overture** framework is a collection of C++ libraries that provide tools for solving partial differential equations. **Overture** can be used to solve problems in complicated, moving geometries using the method of overlapping grids (also known as overset or Chimera grids). **Overture** includes support for geometry, grid generation, difference operators, boundary conditions, database access and graphics. More information about **Overture** can be found at: `www.llnl.gov/casc/Overture` Nothing in **Overture** is in any way tailored for use with a preprocessor.

The main class categories that make up **Overture** are as follows:

- **Arrays** [10]: describe multidimensional arrays using A++/P++. A++ provides the serial array objects, and P++ provides the distribution and interpretation of communication required for their data parallel execution.
- **Mappings** [5]: define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.
- **Grids** [2, 4]: define a discrete representation of a mapping or mappings. These include single grids, and collections of grids; in particular composite overlapping grids.
- **Grid functions** [4]: storage of solution values, such as density, velocity, pressure, defined at each point on the grid(s). Grid functions are derived from A++/P++ array objects.
- **Operators** [1, 3]: provide discrete representations of differential operators and boundary conditions
- **Grid generation** [6]: the Ogen overlapping grid generator automatically constructs an overlapping grid given the component grids.

- **Plotting** [7]: a high-level interface based on OpenGL allows for plotting **Overture** objects.
- **Adaptive mesh refinement:** The AMR++ library is an object-oriented library providing patch based adaptive mesh refinement capabilities within **Overture** .

Object-oriented abstractions are present at many levels within **Overture**, but within this paper we focus on the lower level array objects and array operators. Numerous mechanisms have been implemented previously to optimize the performance of the A++P++ array class library, these mechanisms have included highly optimized binary operator[21], deferred evaluation[15], and expression templates[23, 18, 19].

2.1 Array Abstractions

A++ and P++ [10, 21] are array class libraries for performing array operations in C++ in serial and parallel environments, respectively.

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction specifically well suited to large-scale numerical computation. It provides efficient use of multidimensional array objects which serves to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, effectively allowing A++ serial applications to be recompiled using P++ and thus run in parallel. This provides a simple and elegant mechanism that allows serial code to be reused in the parallel environment.

P++ provides a data parallel implementation of the array syntax represented by the A++ array class library. To this extent it shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model. However, in contrast to HPF, P++ provides a more general mechanism for the distribution of arrays and greater control as required for the multiple grid applications represented by both the overlapping grid model and the adaptive mesh refinement (AMR) model.

Here is a simple example code segment that solves Poisson's equation in either a serial or parallel environment using the A++/P++ classes. Notice how the Jacobi iteration for the entire array can be written in one statement.

```
// Solve u_xx + u_yy = f by a Jacobi Iteration
   Range R(0,n)                    // a range of indices: 0,1,2,...,n
   floatArray u(R,R), f(R,R)       // declare two two-dimensional arrays
   f = 1.; u = 0.; h = 1./n;       // initialize arrays and parameters
   Range I(1,n-1), J(1,n-1);       // define ranges for the interior

// data parallel statement
   for( int iteration=0; iteration<100; iteration++ )
      u(I,J) = .25*(u(I+1,J)+u(I-1,J)+u(I,J+1)+u(I,J-1)-f(I,J)*(h*h));
```

This example shows the array abstractions in use, the optimization of these sorts of statements (and many more complex) are a driving interest in the development of ROSE as an optimization mechanism. Here, the array objects are defined with overloaded operators for +,−,(),∗, and =. The resulting execution is pairwise if binary operators are used, and a more efficient if expression templates are used. But many targets of optimization involve multiple statements where the expression template mechanism only provides for single statement optimizations. Cache based transformations have also been worked out that can (while complex) surpass fortran 77 performance by a factor of four! The development of preprocessors using ROSE should allow the automated introduction of these much more sophisticated transformations. Currently, only less sophisticated transformations have been automated using preprocessors built using ROSE.

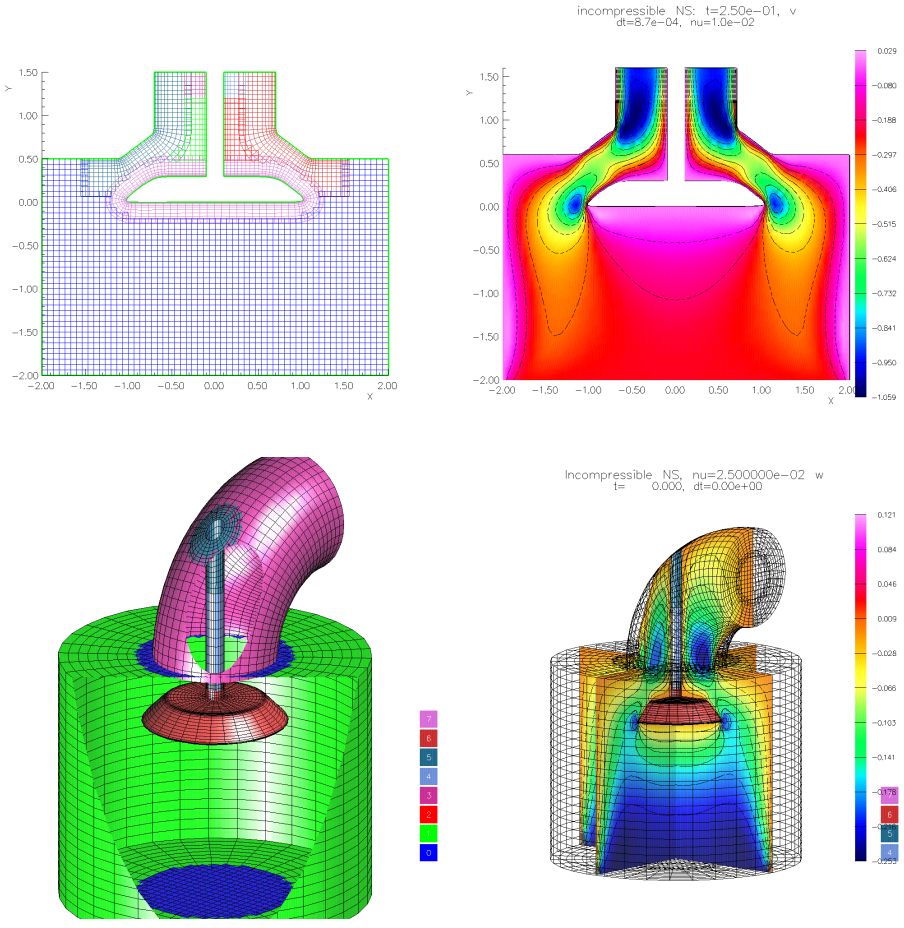

Fig. 1. Sample 2D and 3D **Overture** overlapping grids and applications.

2.2 Grid-function Abstractions

This example demonstrates the power of the **Overture** framework by showing a basically complete code that solves the partial differential equation (PDE)

$$u_t + au_x + bu_y = \nu(u_{xx} + u_{yy})$$

on an overlapping grid. This example shows the higher level abstractions represented within **Overture** (beyond that of the array abstractions). The interaction of these abstractions is what we seek to optimize using the preprocessors built from ROSE.

```
int main()
\{
  CompositeGrid cg;                          // create a composite grid
  getFromADataBaseFile(cg,"myGrid.hdf");     // read the grid in
  floatCompositeGridFunction u(cg);          // create a grid function
  u=1.;                                      // assign initial conditions
  CompositeGridOperators op(cg);             // create operators
  u.setOperators(cg);
  PlotStuff ps;                              // make an object for plotting
  // --- solve a PDE ----
  float t=0, dt=.005, a=1., b=1., nu=.1;
  for( int step=0; step<100; step++ )
  \{
    u+=dt*( -a*u.x()-b*u.y()+nu*(u.xx()+u.yy()) );
    t+=dt;
    u.interpolate();                 // interpolate overlapping boundaries
    // apply the BC u=0 on all boundaries
    u.applyBoundaryCondition(0,dirichlet,allBoundaries,0.);
    u.finishBoundaryConditions();
    ps.contour(u);                    // plot contours of the solution
  \}
  return 0;
\}
```

This example solves the time-dependent equation explicitly. Other class libraries within the **Overture** framework simplify the solution of elliptic and parabolic equations, the linear systems generated can be solved using any of numerous numerical methods as appropriate including multigrid, and methods made available within a number of external dense and sparse linear algebra packages. Figure 1 shows the sorts grids and example applications possible with **Overture** .

The array and grid-function abstractions demonstrated can be significantly optimized over the execution provided by the C++ implementation directly. As implemented, via a library, the library can do little to optimize the execution since it can not see the context of the full statement (or surrounding statements).

Fundamentally, using the semantics of these abstractions to drive optimizations is easier and more productive than relying upon classical program analysis

of the much lower level and more general abstractions of the C++ language itself. While the C++ language's abstractions are general and flexible to allow general use, an object-oriented framework's abstractions can be made arbitrarily precise. The resulting semantic knowledge can be much greater than that provided through program analysis. In general, it is more likely that a combination of leveraging semantics and less sophisticated program analysis will provide a more complete solution.

3 Design of Preprocessors

To encapsulate the semantics of an object-oriented framework's abstraction into the compilation process we must identify the uses of the abstraction within a user's application code. The naive approach to this is:

1. traverse the abstract syntax tree (AST) represented by an application,
2. search for types,
3. put together their use relative to one another.

While simple to explain superficially, the mechanism is particularly complex due to the depth of C++ syntax behind which many high-level abstractions, and the interactions between them, are buried within the AST. The shear size of the program tree for realistic scientific applications further complicates this approach.

To clarify the representation of high-level abstractions, and more importantly their interactions (which can be particularly complex), we simplify the AST to an intermediate form where both the use of a framework's abstractions and their interactions can be more immediately recognized (the simpler the better). This intermediate form is customized to a framework's abstractions.

To represent a framework's abstractions in a higher level representation (an alternative intermediate form) of the user's application's program tree (or more specifically an alternate AST) we build a grammar specific to the framework's abstractions. This higher-level grammar is then used to parse the base level (C++) AST. The result is an intermediate form tailored to an object-oriented framework's abstractions and a new AST for a user's application. The new AST using the high-level grammar dramatically simplifies the recognition of a framework's abstractions, and the interactions between them, within the user's application. This new higher-level AST can be more readily traversed and a framework's abstractions and interactions between abstractions more readily identified.

Since the development of the higher level grammars specific to an object-oriented framework could be rather complex we have automated this process. The development of the parser from the C++ AST to the higher-level grammar's AST is also automated. The result is a mechanism that automatically generates the significant pieces required to form a preprocessor for an arbitrary object-oriented framework.

The preprocessing phase for an application code involves two phases:

1. Recognition of "where" to apply transformations
   The details of this phase are the subject of this paper.
2. What transformation to apply
   We show results of the use of a preprocessor formed using ROSE (in section 5) with a simple transformation to improve the performance of the A++P++ array class library within the **Overture** framework. This paper does not go into any details regarding this specific transformation which in this case basically provides for an automated lowering of the code which would execute the C++ object interactions (array expressions) to C code which is more efficient (and more architecture specific).

The *generation* of the preprocessor involves more steps and is the focus of this paper. Of the two phases, the difficult aspects of the preprocessor are mostly related to the recognition phase:

1. Building the base level C++ grammar and their object-oriented implementation
2. Building the higher level application specific grammars and their object-oriented implementation
3. Parsing the C++ application into the Base level C++ grammar
4. Parsing the C++ application into the higher-level application specific grammars

A *Meta-program* level is used to define the preprocessor, this level is a simple C++ application code. The *Meta-program* defines the manipulation of grammars using the **ROSETTA** library. The output of the *Meta-program*, when it is executed, is source code (written to files). The source code is compiled, with the ROSE infrastructure, to form a preprocessor specific to a given framework. The *Meta-program* can 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 need not be recompiled by the user.

4 ROSETTA

ROSETTA is a tool we developed for the manipulation of grammars. It permits a C++ *Meta-program* to be defined which, when executed, builds tools like Sage II. It is **not** a novel part of this work to have defined a mechanism to generate the Sage II source, modified or not. It **is** a novel part of this research work that higher-level grammars can be automatically generated in addition to the Sage II source. This important feature is the mechanism by which critical parts of the preprocessor are customized for a framework's abstractions; and automatically generated.

ROSETTA represents a class library of terminals and nonterminals used to define a grammar. It is relatively trivial to define the C++ grammar in terms of terminals and nonterminals and associate with the terminals and nonterminals application code. We consider an implementation of the grammar to be a library

of classes representing the different language elements defined by a grammar (statements, expressions, types, etc.). We use the Sage II library as a basis for our C++ grammar, but other libraries that implement grammars and form the basis of different sorts of compiler tools exist[11, 8].

4.1 Generation of the C++ Grammar's Implementation

The *Meta-program* for the construction of the modified version of Sage II that we build is just:

```
// include definitions of grammars, terminals, and non-terminals
// (objects within ROSETTA)
#include "grammar.h"

int main()
   {
  // Build the C++ grammar (generate Sage II source)
     Grammar sageGrammar;

  // Build the header files and source files
  // representing the grammar's implementation
     sageGrammar.buildCode();
   }
```

Here the example program builds the implementation of the C++ grammar (mostly represented as a copy of the Sage II source code with modifications). The output of this application is about 70,000 lines of source code. With the output files compiled into a preprocessor and linked with the ROSE infrastructure, the final preprocessor parses C++ applications and unparses them to generate C++ (identical to the input code in format as well as syntacticly). Such a preprocessor is of little use for our purposes but forms a trivial example of a preprocessor built using ROSE.

4.2 Generation of a High Level Grammar's Implementation

This section explains the system of constraints used to define higher level grammars (higher level and more specific than the C++ grammar). The principle is to include and exclude terminals in an existing grammar (the Base grammar for our purposes is the C++ grammar). Terminals are added or removed as desired to define modifications of the C++ grammar. As an example, additional terminals can be added to define additional types represented by a class defined within an object-oriented framework. New terminals are added through the specification of an existing C++ terminal *plus* constraints. The form of the constraints can be varied (and are expressed using C++ code).

As an example, the specification of a class name could be used to define a new terminal in a new grammar specific to a class name associated with a framework's abstraction (assuming the abstraction is an object). The result is a grammar for which the framework's abstraction is recognized as an implicit type within the

higher-level grammar. The use of the framework's abstraction within expressions can be recognized through the addition of expression terminals to the higher-level grammar. Since all elements of the higher-level grammar are built from terminals of the C++ grammar with an additional constraint no modifications to the C++ language are possible. This is a strength of this mechanism since we want to recognize a framework's abstractions and not formally extend the C++ language.

To further customize the high-level grammar to a particular framework's abstractions, the addition of a new type terminal drives the automated introduction of all possible expression terminals with the constraint that they are between objects of the new added type. The classes represented by the new types are further interrogated to define all possible expressions (member functions of the framework's abstraction) represented by the new type. Similarly statement terminals are added to represent statements containing expressions in the new type. Since the addition of new types adds to the number of terminals (and non terminals) in a grammar, the size of the grammar's implementation nearly doubles. Since this step is fully automated, the amount of additional code generated is not important. Within this approach, through the design of the higher level grammars, we permit user defined types and their expressions and statements to be treated as implicit keywords within an user's application.

5 Preprocessing Overture Applications

The execution of array statements involves inefficiencies stemming from several sources and the problem has been well documented, by many researchers[23, 18, 19]. Having tried all previously conceivable approaches, our approach to performance within **Overture** is to use a preprocessor to introduce optimizing source-to-source transformations. The C++ source-to-source preprocessor is built using ROSE.

The preprocessor built using ROSE has a few features that stand out:

1. A hierarchy of grammars are specified as input to ROSE to build (tailor) the preprocessor specific to a given object-oriented application, library, or framework. ROSETTA is used to generate an implementation of the grammars that are used internally. The hierarchy of grammars (and their implementations) are used to construct separate program trees internally, one program tree per grammar, each representing the user's application. The program trees are edited as required to replace selected subtrees with other subtrees representing a specific transformation. Quite complex criteria may be used to identify where transformations may be applied, this mechanism is superior to pattern-recognition of static subtrees within the program tree because it is more general, readily tailored, and far easier to use.
2. Transformations are specified which are then built into the user application automatically where appropriate. The mechanism is designed to permit the automated introduction of particularly complex transformations (such as

the cache based transformations specified in [20], space does not permit an elaboration of this.

3. To simplify the debugging, the preprocessor's output (C++ code) is formatted identical to the input application code (except for transformations that are introduced, which have a default formatting). Numerous options are included to tailor the formatting of the output code and to simplify working with either its view directly within the debugger or its reference to the original application source within the debugger. Comments and all C preprocessor (cpp) control structures are preserved within the output C++ code.
4. The design of ROSE is simplified by leveraging both Sage II and the EDG[14] C++ front-end. EDG supplies numerous vendors with the C++ front-end for their compiler and represents the current best implementation of C++. In principle this permits the preprocessors built by ROSE to address the complete C++ language (as implemented by the best available front-end). Modifications have been made to Sage II to permit portability and allow us to fulfill on a complete representation of the language. By design, we leverage many low-level optimizations provided within modern compilers while focusing on higher level optimizations largely out of reach because traditional approaches can not leverage the semantics of high level abstractions. In doing so, we slightly blur the distinction between a library or framework, a language, and a compiler. But, because we leverage several good quality tools the implementation is greatly simplified.

5.1 Results

Within our results we consider the following trivial five-point stencil:

```
A(I,J) =  c * ( 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 (distributed across multiple processors if P++ is used). In this example, `I` and `J` are `Range` objects that together specify an index space of the arrays `A` and `B`.

Figure 2 shows the range of performance associated with different size arrays for the simple five point stencil operator on the Sun Ultra and Dec Alpha machines. The Sun Ultra was selected because it is a commonly available computer, the Dec Alpha was selected because its cache design is particularly aggressive and as a result it exemplifies the hardest machine to get good cache performance. The results are in no way specific to this statement, moderate size applications have been processed using preprocessors built with ROSE. The results compare the ratios of A++ performance with and without the use of the ROSE preprocessor to that of optimized C code. The optimized C code takes full advantage of the bases of the arrays being identical and the unit strides, the A++ implementation does not, these very general subscript computations within the array

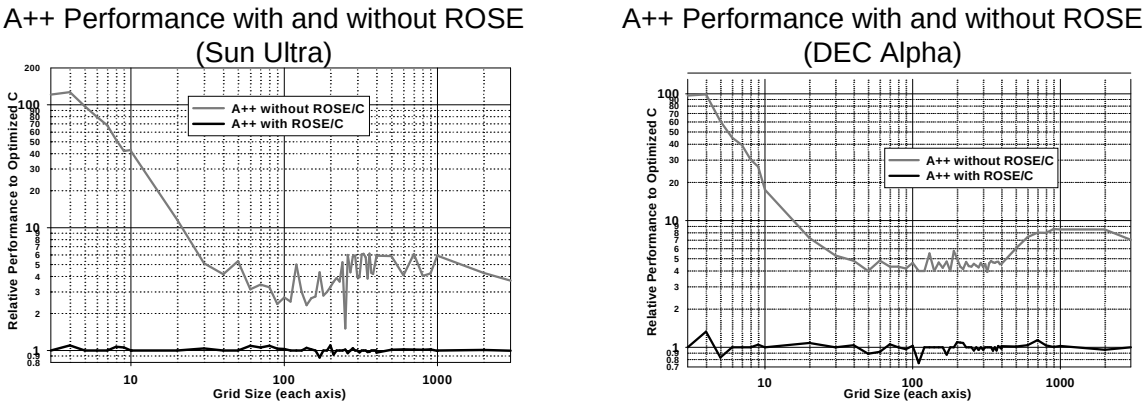

Fig. 2. The use of a preprocessor (built using ROSE) can overcome the performance degradation associated with binary evaluation of array operands. These results show the use of ROSE with A++ and how the performance matches that of optimized C code using the `restrict` keyword (ratio = 1). It has been shown previously that this is equal to Fortran 77 performance. More sophisticated cache-based transformations are also possible.

class implementation are compared to very specific and highly optimized subscript computations within the C code. This exaggerates the poorer performance of the A++ statements, we do this to make clear that the performance of the code output from the ROSE preprocessor is in fact highly optimized and is made specific to the common bases of the operands (determined at compile time) and the unit stride (determined at runtime). Our results show the relative difference that it makes to compare such results. The resulting performance using ROSE is nearly identical to that of the optimized C code (ratio = 1), this is not surprising since the preprocessor transformation replaces the array statement with the equivalent C code (highly optimized, and using restrict pointers where they are supported).

A++ supports expression templates but this data is not presented here, in general the expression template will approach the C performance within 90% for short expressions and sufficiently large arrays. The combination of expression templates with deferred evaluation reduces this to approx. 70% as reported in [19] likely because of the required extra level of indirection to the data required by the deferred evaluation mechanism (it is not clear if this will be fixed)[1].

An important distinguishing point between the two approaches is that within larger applications the compile times are several orders of magnitude less for the preprocessor approach since expression templates are not used[23]. In practice the time taken to pre-process an application is even much less than the compile

[1] This was the experience with expression templates when it was combined with the deferred evaluation mechanism in A++P++.

time where no templates are used (expression templates or otherwise) (a few seconds, and is not noticeable). This is not surprising since the preprocessing consists of only a few of the steps taken internally within a compiler, and excludes the most time consuming back-end optimization (to build the object code).

6 Conclusions

Overture is capable of addressing the complexity of numerous difficult sorts of simulations within scientific computing. While the abstractions presented within **Overture** are the principle motivation for its use, the performance of **Overture** is critical and is dominated by the performance of the A++P++ array class. Many years of work have gone into the development of optimization techniques for the array class library. The preprocessor approach is by far the most successful so far, however more work remains to make preprocessors easier to build and more robust.

The approach within ROSE is different from other open C++ compiler approaches because it provides a mechanism for defining high level grammars specific to an object-oriented framework and a relatively simple approach to the specification of large and complex transformations. A requirement for representing the program tree within different user defined grammars is to have access to the full program tree, this is not possible (as we best understand) within the OpenC++[12] research work. By using Sage II and ROSE the entire program tree, represented in each grammar, is made available; this permits more sophisticated program analysis (when combined with the greater semantic knowledge of object-oriented abstractions) and more complex transformations. We believe that the techniques we have developed greatly complement the approaches represented within OpenC++, in particular the Meta object mechanism represented within that work. That Sage II is in many ways similar to the MPC++[11] work, we believe we could have alternatively built off of that tool in place of Sage II (though this is not clear). However, since Sage II uses the EDG front-end we expect this will simplify access to the complete C++ language. MPC++ addresses more of the issues associated with easily introducing some transformations than Sage II, but not of the complexity that we require for cache based transformations[20]. Each represent only a single grammar (the C++ grammar) and this is far too complex (we believe) a starting point for the identification of where sophisticated transformations can be introduced. The overall compile-time optimization goals are related to ideas put forward by Ian Angus[17], but with numerous distinguishing points:

1. We have decoupled the optimization from the back-end compiler to simplify the design.
2. We have developed hierarchies of grammars to permit arbitrarily high level abstractions to be represented with the greatest simplicity within the program tree. The use of multiple program trees (one for each grammar) serves to organize high level transformations.

3. We provide a simple mechanism to implement transformations.
4. We leverage the semantics of the abstractions to drive optimizations.
5. We have implemented and demonstrated the preprocessor approach on several large numerical applications.

Finally, because ROSE is based ultimately (through Sage II) upon the EDG C++ front-end, the full language is made available; consistent with the best of the commercial vendor C++ compilers which most often use the same EDG C++ front-end internally. However, some aspects of the complete support of C++ within Sage II are incomplete (hence our modifications to fix these details). This in now way makes the Sage II work any less impressive and we are thankful for the use of Sage II.

The results we have presented demonstrate the optimization of array class statements. All sizes of arrays benefit, their processing with ROSE makes each equivalent to the performance of optimized C code (using restrict). Previously in [22] we showed that this is equivalent to FORTRAN 77 performance.

Expression Templates is an alternative mechanism that can be used to optimize array statements, but the mechanism is problematic[23]. More research is required (and being done by others) to address problems within the expression template mechanism. More work is similarly required to provide improved compile-time optimization solutions.

References

1. ———, *Classes for finite volume operators and projection operators*, LANL unclassified report 96-3470, Los Alamos National Laboratory, 1996.
2. G. S. Chesshire, **Overture** *: the grid classes*, LANL unclassified report 96-3708, Los Alamos National Laboratory, 1996.
3. ———, *Finite difference operators and boundary conditions for Overture, user guide, version 1.00*, LANL unclassified report 96-3467, Los Alamos National Laboratory, 1996.
4. ———, *Grid, GridFunction and Interpolant classes for* **Overture** *, AMR++ and CMPGRD, user guide, version 1.00*, LANL unclassified report 96-3464, Los Alamos National Laboratory, 1996.
5. ———, *Mappings for* **Overture** *: A description of the mapping class and documentation for many useful mappings*, LANL unclassified report 96-3469, Los Alamos National Laboratory, 1996.
6. ———, *Ogen: an overlapping grid generator for Overture*, LANL unclassified report 96-3466, Los Alamos National Laboratory, 1996.
7. ———, *PlotStuff: a class for plotting stuff from* **Overture** , LANL unclassified report 96-3893, Los Alamos National Laboratory, 1996.
8. Georges-Andre Silber, `http://www.ens-lyon.fr/ gsilber/nestor/index.html.`
9. D. Quinlan, *Adaptive Mesh Refinement for Distributed Parallel Processors*, PhD thesis, University of Colorado, Denver, June 1993.
10. ———, *A++/P++ manual*, LANL Unclassified Report 95-3273, Los Alamos National Laboratory, 1995.

11. 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
12. 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
13. B. Francois 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.
14. Edison Design Group http://www.edg.com
15. Rebecca Parsons and Dan Quinlan. *A++/P++ array classes for architecture independent finite difference computations.* In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OONSKI'94)*, April 1994.
16. Dan Quinlan and Rebecca Parsons. *Run-time recognition of task parallelism within the P++ parallel array class library.* In *Proceedings of the Conference on Parallel Scalable Libraries*, 1993.
17. Ian Angus *Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More.* In *Proceedings of the Object-Oriented Numerics Conference*, (OONSKI) 1993
18. Todd Veldhuizen *Arrays in Blitz++* In *Proceeding of the Second International Symposium, ISCOPE 98*, Santa Fe, NM December 1998
19. 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
20. 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
21. 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.
22. 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
23. 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