ROSE: A General Tool for the Independent Optimization of Object-Oriented Frameworks

Kei Davis[1], Bobby Philip[2], and Dan Quinlan[3]

[1] Los Alamos National Laboratory, Los Alamos, NM 87545, USA,
kei@lanl.gov,
WWW home page: http://www.c3.lanl.gov/~kei/
[2] University of Colorado at Boulder, Boulder, CO, USA,
bobbyp@colorado.edu,
WWW home page: http://ucsu.colorado.edu/ bobbyp/bhom.html
[3] Lawrence Livermore National Laboratory, Livermore, CA, USA,
dquinlan@llnl.gov,
WWW home page: http://www.llnl.gov/CASC/people/quinlan/

Abstract. ROSE is a programmable source-to-source transformation tool for the optimization of C++ object-oriented frameworks. Because it is programmable, explicit knowledge of framework semantics may be exploited; in contrast the potential capability of a general-purpose compiler is limited by computable semantic inference. Since ROSE is programmable, additional specialized program analysis is possible using implicit knowledge of the workings of the framework, for example, dependence analysis at the level of the framework's abstractions. This enables far greater optimization than is even theoretically possible by a general-purpose compiler. ROSE specifically responds to the realization that to achieve acceptable performance, in general it is insufficient to optimize a framework; its *use* must also be optimized.
The interface to ROSE is particularly simple and takes advantage of standard compiler technology. ROSE acts like a preprocessor, since it takes as input and produces as output standard C++[1]. Its use is always optional since it is not intended to change the denotational semantics (as opposed to the operational or resource-usage semantics). It cannot be used to introduce *any* new language features or syntax. Importantly, since ROSE generates C++ code, its use does not preclude the use of other tools or mechanisms that would work with an application source code (including class template mechanisms).

1 Introduction

The development of object-oriented frameworks represents the centralization of expertise and its reuse by numerous people, research groups, institutions, and industries. The expertise embodied by object-oriented frameworks ranges widely, and of interest to us includes support for complex geometries and grid

[1] ISO/IEC 14882:1998 C++ standard as implemented by the Edison Design Group

generation, the encoding of advanced numerical algorithms (such as adaptive mesh refinement), and the encapsulation of parallelism on advanced computer architectures. Lawrence Livermore National Laboratory's (LLNL) *Overture* [2] framework has been applied within several disciplines including computational biology at UC Davis, modeling and design of sails for the America's Cup Y-acht Racing at Doyle, and the design of diesel engine simulations at LLNL in collaboration with Caterpillar Inc. Other frameworks have likely also received broad use spanning multiple research and industrial disciplines. The remarkable breadth of different areas of expertise represented by individual object-oriented frameworks has not been entirely offset by the performance issues associated with high performance computing, particularly at national laboratories where high performance within computational simulations is of great concern (and the parallel computer architectures more specialized, complex, and obscure).

Because the rich semantics of object-oriented scientific frameworks are implemented in the relatively primitive and unconstrained language C++, the problem of 'deep' program optimizations are intractable, uncomputable, or impractical (this lattermost in part because of the separate compilation problem). The use of *expression templates* [12] has demonstrated the use of the C++ template mechanism to introduce statement-level transformations, in particular for array-class libraries. Such optimizations can be useful but are fundamentally limited because they cannot encode program analysis. The more classic use of binary operators for the separate pairwise evaluation of expressions has similar can encode run-time, but not compile-time, analysis of a limited sort [9].

As a very simple example, without optimization the expression `A=B+C+D+E` denoting the addition of four arrays and assignment of the result entails the creation and destruction of three intermediate arrays and copying to the fourth, and at least four major loops. Not only is this inefficient in itself, but for cache-based architectures the effect on performance can be disastrous. In contrast, it has been demostrated that transformations to exploit cache can result in performance 3-4 times *greater* than straightforward implementation in FORTRAN 77 or C [1, 7, 6]. This example also makes clear that it is not sufficient to optimize the framework or library itself; its *use* must also be optimized: while some cache optimization could conceivably be encoded in the framework, it is the context of the objects that must be analyzed ot make general cache-based transformations possible and worthwhile.

ROSE was conceived as a general mechanism for implementing program transformations to remove such sources of inefficiency. Its strength is its complete programmability. In practice the conceptual tradeoff is to forego deep program analysis in favor of more shallow analysis and some encoding of the semantics of the target framework.

Serial optimizations are only the first step for parallel architectures. Parallel optimizations are also possible (for example, scheduling of communication), but not by a compiler for an inherently sequential language such as C++. ROSE provides a mechanism for performing parallel optimizations in this context.

1.1 Scope

At its simplest ROSE is a tool for performing arbitrary source-level transformations to C++ programs. In practice ROSE provides functionality to greatly simplify the implementation of transformations within applications using object-oriented abstractions implemented by an object-oriented framework. Our primary work has been on optimizations for A++/P++ array classes within the Overture framework, but this has been shown to be readily applicable to other array class libraries. Current target optimizations include:

- loop fusion of (multiple) binary operators;
- loop fusion across statement (requiring dependence analysis);
- cache-based optimizations;
- temporal locality optimizations;
- introduction of performance-gathering options and metrics.

More specifically, the targeted object-oriented frameworks/libraries are

- A++/P++ (in OVERTURE, from LLNL) [8];
- POOMA (from LANL) [4];
- Blitz (from University of Waterloo) [11];
- GNU SSL (from the Free Software Foundation) [5];
- ValArray (from the C++ Standard Library) [10];
- a 'least common denominator' array class library.

OVERTURE and POOMA comprise considerably more than an array class library; it is the array-class library subsets that are of interest; for POOMA it is the temporal-locality and cache-based optimizations that are relevant.

2 How ROSE Works

ROSE is 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 allow critical dependence on the optimization step.

ROSE is built on the Sage II source code restructuring tool from University of Indiana and ISI [3]. Sage II 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 (actually a graph).

By way of running example, in the following we will consider only a single transformation of a simple 1D array statement. The specification of a transformation consists of two parts:

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

2.1 Specification of where to introduce optimizations

Recognition of syntax subject to transformation is automatic, that is, program annotations (such as pragmas) are not used. except as a mechanism for disabling transformation in specific segments of code for evaluation purposes. Such syntax is specified using a formal grammar using conventional extended BNF notation, augmented by type information. This is encoded in tabular form to make extension and modification of the grammar simple.

2.2 How the grammar is defined

In the example of a transformation of array assignment statements two grammars will be defined. The first is a grammar defined for the array class. Space does not permit the presentation of this grammar it its entirety (it will be made available at the Overture WWW site)—Figure 1 shows a subset of the grammar defining the A++/P++ array class library syntax (the *array grammar*). Higher level grammars specific to each transformation can be defined in terms of the array grammar. Figure 2 shows the *array assignment grammar*. More complex transformations (for stencil operations, for example) use yet higher level grammars defined in terms of the array assignment grammar.

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

Fig. 1. Example of product rule for nonterminals of the *array grammar* using the mechanisms for defining grammars within ROSE.

2.3 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-based implementation of the grammar used to represent a program tree using the associated grammar. For large grammars this mechanism obviates the need for hand coding many thousands of lines of code that would be required to represent the implementation of the grammar. Such large grammars can be expected for sophisticated object-oriented frameworks (though subsets of a framework can be targeted to reduce the size of the grammar). The user interface of ROSE permits customization of the behavior of these implementations of the grammar, allowing user-defined code to be inserted into the implementations of the grammars.

```
Grammar::NonTerminal arrayOperator = arrayBinaryOperator;
Grammar::NonTerminal assignmentOperator = arrayAssignmentOperator;
Grammar::NonTerminal transformableExpression;
transformable_expression =
  transformableExpression & operator & transformableExpression |
  arrayOperand arrayOperator arrayOperand |
  arrayOperand;
Grammar::NonTerminal lhs_operand = arrayOperand;
Grammar::NonTerminal rhs_operand = transformable_expression;
Grammar::NonTerminal transform_statement =
  lhs_operand & assignmentOperator & rhs_operand;
```

Fig. 2. The *array assignment grammar*

One goal of this research is to automate as much as possible the generation of the grammars.

The use of grammars provides a mechanism to differentiate the application code. As a rule, if a sub-tree of the program tree provided by Sage II can be recognized using a grammar then the program tree has a derivation from that grammar. The array grammar is used to recognize array operations (statements, expressions, types, etc.). It is simpler to use a multi-stage approach: successively refining the recognition process by using a sequence of grammars—formally this amounts to successive intersection operations. The specification of the second grammar is used to identify array assignment statements, for example. The use of yet another grammar can be used to further refine (filter) the collection of array assignment statements, for example to identify stencil-like operations targeted for cache-based transformations.

2.4 Specification of a Transformation

The specification of a transformation completes the process of defining an optimization. The specification of the transformation must be represented in multiple parts and these parts must be assembled according to the the context of the original statement (in the case of an array assignment optimization the context may include the dimension, number of operands, etc., if this information is not represented in the grammar directly). The transformation of the program tree occurs as a transformation of the program tree associated with a later grammar into a program tree associated with an earlier one. For example, for an array-assignment statement the transformation consists of a transformation of the syntax tree associated with the array-assignment grammar into a tree associated with the array grammar, then to a tree associated with the array grammar.

For each element of the grammar a transform function is defined (automatic generation of these functions from the definition of the grammar directly from the specification of the transformation rules is planned). This transformation is defined by 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: The transformation is explicitly defined by supplementing the definition of that element of the grammar. Space limitations precludes an example, but required elemets are assembled explicitly from the C++ grammar defined by the Sage II objects.
2. By pattern The transformation is separated into pieces and how the pieces are fit together is defined explicitly.

More mechanisms may be defined as automation is improved.

Transform functions are mutually recursive in a pattern parallel to that of the corresponding grammar.

2.5 Transformation rules

The transformation of the representation of a piece of syntax in one grammar to its representation in another (as a syntax tree) is defined by a set of transformation rules. These rules depend only on the definitions of the two grammars. Currently these rules are generated by hand; research is underway to automate their generation.

Figure 3 shows the specification of the elements of the transformation; space limitations preclude the presentation of the code fragement that shows how the elements representing the transformation are assembled.

As a final example we present two tiny codes, one using the array class directly, and the other being the output of ROSE. These are shown in Figures 4 and 5. Current work on the unparsing of the C++ program tree (built by Sage II) provides various options to control the formatting of ROSE output.

2.6 Summary

Figure 6 shows a flowchart of the use of ROSE and the pieces that build the ROSE preprocessor itself. The use of the preprocessor is shown to be optional with the application code being the only input required if no optimization is performed. If optimization is required then the optimizing preprocessor specific to that framework is used. The preprocessor specific to a given framework is build from the specification of a hierarchy of grammars and associated transformation definitions, together with the ROSE infrastructure.

Target recognition is via an arbitrary sequence of grammars; target transformation is via transformation rules defined in terms of these grammars. Use of such hierarchies is for practical reasons: the total size of multiple grammars is much less than would be single one, and the resulting factoring leads to reuse. The example presented in this paper shows parts of how an array grammar may be specified and how a grammar specific to the optimization of array assignment statements (the array assignment grammar) is defined using that array grammar.

```
FUNCTION_DEFINITION UNIQUE_PART_OF_TRANSFORMATION () {
  // This code is required once for all the operands in the same scope
  int INDEX = 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 each rhs 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 (INDEX = base_1D_0; INDEX <= bound_1D_0; INDEX++) {
    TRANSFORMED_STATEMENT();
  }
}
```

Fig. 3. Array assignment transformation rules.

3 Conclusion

ROSE is a fully programmable tool that provides the capability of arbitrary transformation of C++ code. The intention is that such transformations be semantics-preserving so that its use is always optional, and our goal is aggressive program optimization and the introduction of parallelism with the abstractions introduced by high-level scientific frameworks as the targets. ROSE specifically addresses the realization that to achieve acceptable performance from C++ object-oriented frameworks their use, not just the frameworks themselves, must be optimized.

A driving goal in the development of ROSE is to provide a simple coherent mechanism that makes the task of implementing a particular optimization just a few hours' work. Since different frameworks focus on different features and optimizations, often greatly complicating their implementation; this work will essentially make optimizations available more universally independent of the peculuarities of the framework. By removing much of the optimization from being implemented within the framework, through complex runtime mechanisms, object-oriented frameworks can be greatly simplified and focused upon the *important* abstractions (e.g. grid generation, moving grids, particles, adaptive mesh refinement (AMR), equation solvers, parallel distribution mechanisms, load bal-

```
#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;
}
```

Fig. 4. Example A++ code before processing using ROSE.

ancing, etc.). Additionally, ROSE will permit simplified communication with talented researchers from compiler optimization fields to address highly specialized optimizations. It is hoped that ROSE will effectively level the playing field between different object-oriented frameworks and allow them to focus upon the needs of their specific users more readily.

References

1. Federico Bassetti, Kei Davis, and Dan Quinlan. Toward fortran 77 performance from object-oriented scientific frameworks. In *Proceedings of the High Performance Computing Conference (HPC'98)*, 1998.
2. David Brown, Geoff Chesshire, William Henshaw, and Dan Quinlan. Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, March 1997.
3. 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.
4. J.V.W. Reynders et. al. *POOMA: A Framework for Scientific Simulations on Parallel Architectures*, volume Parallel Programming using C++ by Gregory V. Wilson and Paul Lu, chapter 16, pages 553–594. MIT Press, 1996.
5. Gnu scientific software library. http:// KachinaTech.com.
6. Naraig Manjikian and Tarek Abdelrahman. Array data layout for the reduction of cache conflicts. In *????*, 1997.
7. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
8. 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.
9. 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.

```
#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;}
```

Fig. 5. Example of output from processing of A++ code using ROSE.

10. Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, third edition edition, 1997.
11. Todd Veldhuizen. Blitz++ users manual. http:// monet.uwaterloo.ca/ blitz/ manual/ arrays.html.
12. Todd Veldhuizen. Expression templates. In S.B. Lippmann, editor, *C++ Gems.* Prentice-Hall, 1996.

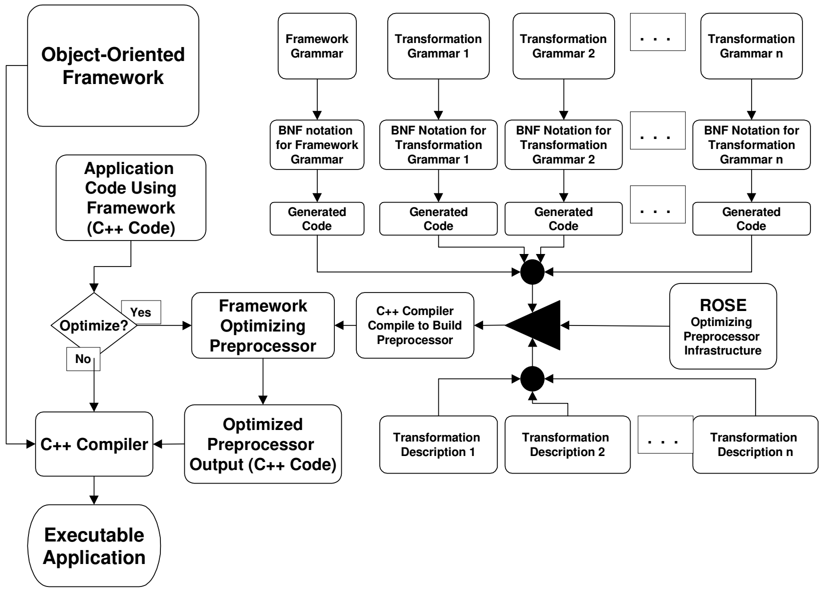

Fig. 6. Flowchart showing how the design and use of ROSE preprocessor infrastructure.