Classification and Utilization of Abstractions for Optimization

Dan Quinlan[1], Markus Schordan[2], Qing Yi[1], and Andreas Saebjornsen[3]

[1]Lawrence Livermore National Laboratory, USA
{dquinlan, yi4}@llnl.gov

[2]Vienna University of Technology
markus@complang.tuwien.ac.at

[3]University of Oslo, Norway
andreas.sabjornsen@fys.uio.no

Abstract. We define a novel approach for optimizing the use of libraries within applications. We propose that library-defined abstractions be annotated with additional semantics to support their automated optimization. By leveraging these additional semantics we enable specialized optimizations of application codes which use library abstractions. We believe that such an approach entails the use of formal methods.
It is a common perception that performance is inversely proportional to the level of abstraction. Our work shows that this is not the case if the additional semantics of library-defined abstractions can be leveraged. We describe ROSE, a framework for building source-to-source translators that perform high-level optimizations on scientific applications. ROSE allows the recognition of library abstractions and the optimization of their use in applications. We show how ROSE can utilize the semantics of user-defined abstractions in libraries within the compile-time optimization of applications.

1 Introduction

User-defined abstractions help software developers be more productive by encapsulating redundant details. Unfortunately the use of these abstractions usually introduces a penalty in performance due to insufficient optimizations from the compilers. We define a performance penalty as the difference between the performance of the direct use of the user-defined abstraction and any semantically equivalent lower level representation. In order to apply optimizations to user-defined abstractions it is often required that these abstractions satisfy certain properties (e.g. operators are side-effect free, array elements are not aliased, etc.). When the compiler cannot verify such properties using program analysis it conservatively disables many performance optimizations. Manually introducing optimization avoid the performance penalty but significantly lower programmer productivity and tie the application to individual computer architectures. Our approach aims at fully automating such optimizations.

We propose to use annotations to bridge the gap between the semantics implied by the compiler and the additional semantics known only by the library developer who implements the user-defined abstractions. We present a compile-time approach for the optimization of scientific applications using the semantics of library abstractions. We use an array abstraction from the A++/P++ array class library [1] as an example to demonstrate our approach toward the general annotation of user-defined abstractions.

Where we have defined a mechanism to optimize the use of a category of abstractions (array abstractions), our goal is to define a more general mechanism to annotate arbitrary abstractions with semantics supporting their optimizations. We have addressed numerous aspects of the more general problem within the current work. While current work supports only the manual introduction of annotations, future work will explore ways to automate the introduction of such annotations. This paper describes work that has been done to develop an annotation based approach to guide the automated optimization of user-defined abstractions, starting with a motivating example.

1.1 Motivating Example

Figure 1 shows an example of a high-level abstraction from the A++/P++ library [1]. It is an array abstraction which is used within current scientific applications and forms an appropriate target for optimization.

The class `Range` defines a one dimensional iteration space with fixed stride. The class `floatArray` defines an array abstraction where `Range` objects serve as parameters to several operators (e.g `+`, `=`, `()`) for iterating over an array. In the example main function, we show how these classes can be used to compute a new value for each point in the array. Using such abstractions programmers can express complex computations without explicitly introducing loops which permits greater flexibility for optimization (e.g. parallelization, layout optimization, communication optimization, etc.). In the example we use a two-dimensional array with type double as element type.

In figure 1, the function `main` consists of four lines. In line #2 two non-aliased `floatArrays` A and B, each of size 100x100, are created. In line #3, two `Range` objects I and J, defining ranges from 1 to 98 with stride 1 are created. In line #4 the computation is defined. For each point within the specified ranges, a new value is computed on the right hand side as the sum of the four neighbors in horizontal and vertical direction. In general, an arbitrary expression can be specified on the right hand side, in particular using the operators available in the class `floatArray`. In this simple example we have restricted the available functions to `+` and `sin`. The full class consists of about 80 different operators.

In order to optimize this computation the compiler must know both the the aliasing patterns between different array objects and the semantics of array operations.We shall show how such properties can be specified using our annotation language in section 4.

In the remaining sections we present the ROSE architecture [2, 3] used for implementing the presented approach, the annotation based mechanism for the

```
class Range {
public:
    Range ( int base, int bound, int stride );
    Range operator+ ( int i );
    Range operator- ( int i );
};
class floatArray {
public:
    floatArray ( int i, int j );
    floatArray & operator= ( const floatArray & X );
    friend floatArray operator+ ( const floatArray & X, const floatArray & Y );
    floatArray operator() ( const Range & I, const Range & J );
    friend floatArray & sin ( const floatArray & X );
};
int main () {                                           /* #1 */
    floatArray A(100,100), B(100,100);                  /* #2 */
    Range I(1,98,1), J(1,98,1);                         /* #3 */
    A(I,J) = B(I-1,J) + B(I+1,J) + B(I,J-1) + B(I,J+1);   /* #4 */
}
```

Fig. 1. Example: Code fragment of header file and program to be optimized.

optimization of array abstractions and performance results showing the significant potential for such optimizations.

2 Architecture

The ROSE infrastructure offers several components to build a source-to-source optimizer. A complete C++ front end is available that generates an object-oriented annotated abstract syntax tree (AST) as an intermediate representation. Optimizations are performed on the AST. Several components can be used to build the mid end: a predefined traversal mechanism, an attribute evaluation mechanism, transformation operators to restructure the AST, and pre-defined optimizations. Support for library annotations is available by analyzing pragmas, comments, or separate annotation files. A C++ back end can be used to unparse the AST and generate C++ code. An overview of the architecture is shown in Fig. 2). Steps 1-7, which can be performed by a ROSE source-to-source optimizer, are described in the following sections.

2.1 Front End

We use the Edison Design Group C++ front end (EDG) [4] to parse C++ programs. The EDG front end generates an AST and performs a full type evaluation of the C++ program. This AST is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree, Sage III, based on Sage II and Sage++[5]. Sage III is used by the mid end as intermediate representation. The annotated library interfaces are header files which are included by the application program. The AST passed to the mid end represents the program and all the header files included by the program (see Fig. 2, step 1 and 2).

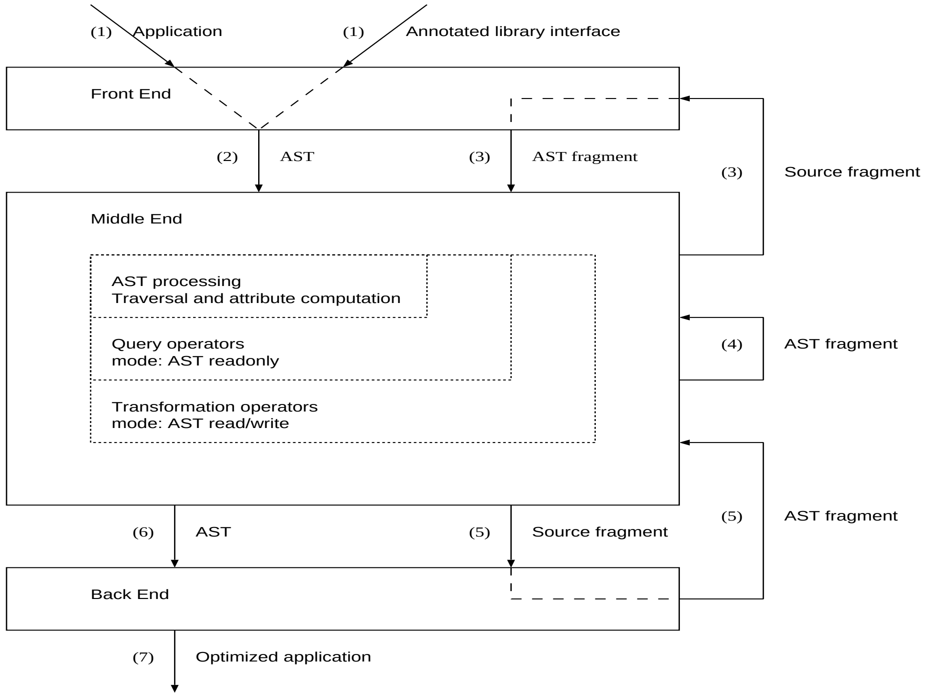

Fig. 2. ROSE Source-To-Source architecture

2.2 Mid End

The mid end permits the restructuring of the AST and performance improving program transformations. Results of program analysis are made available as annotations of AST nodes. The AST processing mechanism computes inherited and synthesized attributes on the AST (see section 2.4 for more details). ROSE also includes a scanner which operates on the token stream of a serialized AST so that parser tools can be used to specify program transformations in semantic actions of an attribute grammar. The grammar is the abstract grammar, generating the set of all ASTs. More details on the use of attribute grammar tools, in particular Coco/R [6] and Frankie Erse's C/C++ port, can be found in [3].

An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. Transformation operators can be built using the AST processing mechanism in combination with AST restructuring operations. In Fig. 2 steps 3,4,5 show how the ROSE architecture also uses source code fragments and AST fragments in the specification of program transformations. A fragment is a concrete piece of code or AST. A program transformation is defined by a sequence of AST restructuring operations whereas transformation operators compute such restructuring sequences. Transformations can be parameterized to define conditional restructuring sequences. This is discussed in detail in section 3.

2.3 Back End

The back end unparses the AST and generates C++ source code (see Fig. 2, steps 6 and 7). It can be specified to unparse either all included (header) files or only the source file(s) specified on the command line. This feature is important when transforming user-defined data types, for example, when adding generated methods. Comments are attached to AST nodes and unparsed by the back end.

2.4 AST Processing

The AST processing components permit traversing the AST and computing attributes for each node of the AST. The computed values of attributes can be attached to AST nodes as annotations and used in subsequent optimizations. Context information can be passed down the AST as inherited attributes and results of computations on a subtree can be computed as synthesized attributes (passing information upwards in the tree). Examples for values of inherited and synthesized attributes are the nesting level of loops, the scopes of associated pragma statements, etc. These annotations can be used in transformations to decide whether a restructuring operation can be applied safely. AST processing is used by query operators and transformation operators to compute information according to the structure of the AST and can also be based on annotations computed by other operators. Using this mechanism library developers can build complex high-level transformation operators from lower-level transformation operators.

2.5 AST Query Operators

Building on top of the methods in section 2.4, AST query operators are provided that perform numerous types of predefined queries on the AST. AST query operators may be composed to define complex queries. This mechanism hides some of the details of the AST traversal and is simple and extensible.

2.6 AST Annotations

Source code annotations are represented within the AST as persistent attributes, which can be directly attached to AST nodes. Such attributes can be accessed by subsequent AST processing steps and permit the communication between different AST processing phases. Annotations within the AST include type information obtained from the EDG front end and user-defined attributes, which specify additional semantic information. The additional information can be utilized in transformations to decide whether a restructuring operation is applicable. Annotations can be introduced using several mechanisms supported within ROSE: pragmas, comments, and a separate annotation file mechanism.

3 Transformation Operators

An optimization requires program analysis to determine whether the AST can be restructured such that the semantics of the program are preserved. For the analysis, the AST processing mechanism allows the computation of attributes. A transformation operator includes a predicate and a sequence of AST restructuring operators. The predicate uses the results of the program analysis and additionally the information provided by annotations. The predicate has to ensure that the restructuring sequence can be applied safely. Therefore we combine conservative program analysis with the additional information from annotations to ensure its correctness. Additionally we also provide operations that enforce a certain transformation, such as the *inline* annotation.

Only if the predicate is true the restructuring operators are applied. The sequence of AST restructuring operations can be computed as attributes by the AST processing mechanism or by using an attribute grammar tool, as demonstrated in [3]. Bottom Up Rewrite Systems (BURS), such as burg [7], can be used to operate on the AST. The AST is implemented such that for each node a unique number is available which can be used as operator identifier by such tools. The opportunity to choose between traversals, the AST processing mechanism, attribute grammar tools, or BURS tools allows selection of the most comprehensive specification of a transformation.

A restructuring sequence consists of fragment operators, and as operands AST fragments (subtrees), strings (concrete pieces of code), or AST locations (denoting nodes in the AST).

3.1 Fragment Operators

A fragment operator transforms a given fragment. It permits performing a basic restructuring operation such as insert, delete, or replace on an AST. The target location of an operation in the AST can be absolute or relative (to another absolute location).

A valid fragment can be specified as source fragment or AST fragment. Whether a source fragment is valid with respect to an absolute location is determined automatically. From the syntactic context of the absolute location the prefix, $s_{\triangleleft}$, is computed such that all declarations, opening scopes, and function headers are included in the prefix. The postfix, $s_{\triangleright}$, consists of all the syntactic entities of closing scopes (for nested scopes such as for-loops, while-loops, function definitions, etc.).

Definition 1 (Valid Source Fragment) *A source fragment $s_{\Box}$ is* valid with respect to an absolute location l_{abs} *in an AST if it can be completed to a legal program from the syntactic and semantic context of the absolute location such that the completed program has correct syntax and semantics, symb. $s_{\Box}$ is valid with respect to l_{abs} if $frontend(s_{\triangleleft} + s_{\Box} + s_{\triangleright})$ with $s_{\triangleleft} = prefix(l_{abs})$, $s_{\triangleright} = postfix(l_{abs})$ succeeds.*

For example, if we want to insert a source fragment into the AST, we specify a target location as an absolute location, l_{abs}, and from that target location the prefix and postfix can be computed to complete the source fragment to a legal program. Then we can invoke the frontend to obtain the translated AST fragment. Thus, for a valid source fragment $s_{\Box}$ we can always generate a corresponding AST fragment, $ast_{\Box}$. The translation of fragments is an extensive operation. For example, in our C++ infrastructure the AST fragment $ast_{\Box}$ has all templates instantiated and types are determined for all expressions.

Operator	Description
insert : $L_{rel} \times L_{abs} \times ASTs \rightarrow ASTs$	Insertion of AST fragment at relative location (step 4 in Fig. 2)
delete : $L_{abs} \times ASTs \rightarrow ASTs$	Deletion of AST subtree at absolute location in AST (step 4 in Fig. 2)
fragment-frontend : $L_{abs} \times ASTs \times S \rightarrow ASTs$	Translate source fragment with respect to absolute location in AST to corresponding AST fragment (steps 3,5 in Fig. 2)
fragment-backend : $L_{abs} \times ASTs \rightarrow S$	Unparse AST fragment at absolute location in AST to source fragment (step 5 in Fig. 2)
locate : $L_{rel} \times L_{abs} \times ASTs \rightarrow L_{abs}$	Map relative location with respect to absolute location in AST to absolute location in same AST
replace : $L_{rel} \times L_{abs} \times ASTs \times ASTs \rightarrow ASTs$	Replacement of AST fragment at relative location (step 4 in Fig. 2)
replace : $L_{abs} \times ASTs \times S \rightarrow ASTs$	Replacement of AST subtree at absolute location in AST by AST fragment corresponding to source fragment (steps 3,4,5 in Fig. 2)

Fig. 3. Fragment operators which allow to modify the AST by using a relative location, an AST fragment, or a source fragment. Transformation operators are defined as sequence of fragment operations.

In Fig. 3 an overview of the most important fragment operators is given. Based on the handling of code fragments, the transformation operators can be defined as follows. Let $ASTs$ denote the set of ASTs, L_{rel} the set of relative locations in an AST, L_{abs} the set of absolute locations, i.e. the nodes in an AST, and S the set of valid source fragments with respect to an absolute location in the AST. The fragment operators allow rewriting the AST by specifying absolute or relative target locations. A relative location l_{rel} allows the specification of a target location in an AST relative to an absolute location l_{abs}. The operator *locate* maps a relative location l_{rel} to an absolute location l_{abs} in a given AST. Relative locations are used to simplify the specification of the target location of a fragment operation. For example, if a statement can be hoisted out of a loop it suffices to specify as target location the “statement in outer scope right before the current loop”. The location “current loop” is specified as absolute location parameter to *locate*. We have defined several classifications of such relative target locations which are useful in making transformations more compact. The insert-operation is an example of using a relative target location. The operator *fragment-frontend* allows translation of source fragments to AST fragments as explained above. It also requires step 5 to compute the necessary prefix and postfix to complete the source fragment to eventually call the front end for the completed program. The unparsing of an AST fragment, *fragment-backend* requires invoking the back end. The last operator listed in Fig. 3, *replace*, allows specification of the new AST fragment, *ast*, which replaces an AST subtree at location L_{abs} in this AST, to be specified by a source fragment, s. This requires all three steps 3,4,5 (see Fig. 2). Step 5 is required to unparse parts of the AST to form the prefix, $s_{\lhd}$, and postfix, $s_{\rhd}$. In Step 3 the completed source fragment is translated to an AST and the corresponding AST fragment, *ast*, is extracted. Step 4 is the actual rewriting of the AST and the replacement of the AST subtree with the new AST fragment is performed. Based on this basic operations on fragments, transformation operators can be defined.

4 Predefined Optimizations

A large set of compiler optimizations, including both reordering transformations such as loop fusion/fission and blocking, and statement level transformations such as redundant expression elimination, can be applied to improve the performance of applications. Most of these optimizations are under certain safety and profitability constraints, which in turn require specific knowledge of the involved operations. However, because user-defined abstractions often introduce function calls with unknown semantics into an application, many of these compiler optimizations are not performed due to the unknown semantics.

In this section we present techniques that extend the applicability of predefined compiler optimizations. By defining an annotation language, which allows programmers to declare that certain abstractions satisfy the extended requirements of predefined compiler optimizations, we provide an open interface for the programmers to communicate with and to control the underlying optimizations. A preliminary version of our annotation language is shown in Figure 4. In the following, we use the annotation examples in Figure 5 to further illustrate the techniques.

4.1 Enabling Transformations

One of the most significant enabling transformations for library abstractions is inlining, which replaces function calls with their implementations within the calling contexts. Suppose the compiler has access to all the source code of a library. Theoretically, inlining the library code could permit the necessary program analysis and thus allow the compiler to discover the semantics of all abstractions, dismissing the concerns for obscure function calls.

However, the current compilation techniques cannot yet fully bridge the gaps between abstraction semantics and their implementation details. Specifically,

```
<annot> ::= <annot1>
    | <annot1>;<annot>
<annot1> ::= class <cls_annot>
    | operator <op_annot>
<cls_annot> ::= <clsname>:<cls_annot1>;
<cls_annot1>::= <cls_annot2>
    | <cls_annot2> <cls_annot1>
<cls_annot2>::= <arr_annot>
    | inheritable <arr_annot>
    | has-value { <val_def> }
<arr_annot>::= is-array{ <arr_def>}
    | is-array{define{<stmts>}<arr_def>}
<op_annot> ::= <opdecl> : <op_annot1> ;
<op_annot1> ::= <op_annot2>
    | <op_annot2> <op_annot1>

<op_annot2> ::= modify <namelist>
    | new-array (<aliaslist>){<arr_def>}
    | modify-array (<name>) {<arr_def>}
    | restrict-value {<val_def_list>}
    | read <namelist>
    | alias <nameGrouplist>
    | allow-alias <nameGrouplist>
    | inline <expression>
<arr_def> ::= <arr_attr_def>
    | <arr_attr_def> <arr_def>
<arr_attr_def> ::= <arr_attr>=<expression>;
<arr_attr> ::= dim  | len (<param>)
    | elem(<paramlist>)
    | reshape(<paramlist>)
<val_def> ::= <name>; | <name>;<val_def>
    | <name> =  <expression> ;
    | <name> =  <expression> ; <val_def>
```

Fig. 4. Annotation Grammar

reading the library code exposes the underlying implementations, but does not readily permit a discovery of the semantics, such as properties of commutativity and associativity. As the result, we complement inlining transformations with semantics annotations which allows library programmers to define the semantics and control the optimizations of their abstractions.

In our annotation language, the programmers can not only customize compilers to inline certain function calls, they can also define additional properties of their abstractions in order to enable specific predefined optimizations. As example, the *inline* annotation in Figure 4 is essentially a "semantics inlining" directive for user-defined functions. In our running example, we use it in the function annotations of class floatArray (see Fig. 5). The function "*floatArray::operator()(int)*" is specified to be inlined and is declared as a subscripted access of the current $floatArray$ object.

4.2 Loop Transformations

As modern computers become increasingly complex, compilers often need to extensively reorder the computation structures of applications to achieve high performance. One important class of such optimizations is the set of loop transformation techniques, such as loop blocking, fusion/fission, and interchange, that has long been applied to Fortran scientific applications. Within ROSE, we have implemented several aggressive loop transformations and have extended them for optimizing loops operating on general object-oriented user abstractions.

Traditional Fortran loop transformation frameworks recognize loops operating on Fortran arrays, that is, arrays with indexed element access and with no aliasing between different elements. After computing the dependence relations between iterations of statements, they then reorder the loop iterations when safe and profitable. To extend this framework, we use an array-abstraction interface to communicate with our loop optimizer the semantics of user-defined array abstractions in C++. The array-abstraction interface both recognizes user-defined array abstractions and determines the aliasing relations between array objects.

```
class floatArray:
    inheritable is-array { dim = 6; len(i) = this.getLength(i); elem(i$x:0:dim-1) = this(i$x);
    reshape(i$x:0:dim-1) = this.resize(i$x); }; has-value { dim; len$x:0,dim-1=this.getLength(x); }
operator floatArray::operator =(const floatArray& that):
    modify {this}; read {that}; alias none;
    modify-array (this) { dim = that.dim; len(i) = that.len(i); elem(i$x:1:dim) = that.elem(i$x); };
operator +(const floatArray& a1,double a2): modify none; read{a1,a2}; alias none;
    new-array () { dim = a1.dim; len(i) = a1.len(i); elem(i$x:1:dim) = a1.elem(i$x)+a2; };
operator floatArray::operator () (const Range& I):
    modify none; read{I}; alias { (result, this) }; restrict-value { this = { dim = 1; };
    result = {dim = 1; len(0) = I.len;}; };
    new-array (this) { dim = 1; len(0) = I.len; elem(i) = this.elem(i*I.stride + I.base); };
operator floatArray::operator() (int index) : inline { this.elem(index) };
    restrict-value { this = { dim = 1; };};
class Range: has-value { stride; base; len; };
operator Range::Range(int _b,int _l,int _s): modify none; read {_b,_l,_s}; alias none;
    restrict-value { this={base =_b;len=_l;stride=_s;};};
operator + (const Range& lhs, int x ) : modify none; read {lhs,x}; alias none;
    restrict-value { result={stride=lhs.stride; len = lhs.len; base = lhs.base + x; };};
```

Fig. 5. Annotations for classes floatArray and Range

In Figure 5, the *is-array* annotation declares that the class $floatArray$ has the pre-defined Fortran array semantics. The array can have at most 6 dimensions, with the length of each dimension i obtained by calling member function *getLength*(i), and with each element of the array accessed through the "()" operator. Here the expression $i\$x : 0 : dim - 1$ denotes a list of parameters, i_1,i_2,...,i_{dim-1}. In the grammar the nonterminal <paramlist> corresponds to such expressions. Similarly, the operator "*floatArray::operator= (const floatArray& that)*" is declared to have *modify-array* semantics; that is, it performs element-wise modification of the current array. The operator "*+(const floatArray& a_1, double a_2)*" is declared to have the *new-array* semantics; that is, it constructs a new array with the same shape as that of a_1, and each element of the new array is the result of adding a_2 to the corresponding element of a_1. Similarly, the operator "*floatArray::operator()(const Range& I)*" constructs a new array that is aliased with the current one by selecting only those elements that are within the iteration range I.

Because the safety of loop optimizations is determined by evaluating the side-effects of statements, our annotation language also includes declarations regarding the side-effects of function calls. Specifically, the *mod* annotation declares a list of locations that might be modified by a function call, the *read* annotation declares the list of locations being used, and the *alias* annotation declares the groups of names that might be aliased to each other. These annotations directly communicate with our global alias and side-effect analysis algorithms.

4.3 Statement level optimizations

To generate efficient code, most compilers eliminate redundant computations or replace expensive computations with cheaper ones. Commonly used optimizations include constant propagation, constant folding, strength reduction, redundant expression elimination, and dead code elimination. Most of these optimizations

```
void interpolate2D ( floatArray & fineGrid, floatArray & coarseGrid )
{
     int fineGridSizeX   = fineGrid.getLength(0);
     int fineGridSizeY   = fineGrid.getLength(1);
     int coarseGridSizeX = coarseGrid.getLength(0);
     int coarseGridSizeY = coarseGrid.getLength(1);
     // Interior fine grid points
     Range If (2,fineGridSizeX-2,2);  Range Jf (2,fineGridSizeY-2,2);
     Range Ic (1,coarseGridSizeX,1);  Range Jc (1,coarseGridSizeY-1,1);
     // separate loops to be fused
     fineGrid(If,Jf)     =  coarseGrid(Ic,Jc);
     fineGrid(If-1,Jf)   = (coarseGrid(Ic-1,Jc) + coarseGrid(Ic,Jc)) / 2.0;
     fineGrid(If,Jf-1)   = (coarseGrid(Ic,Jc-1) + coarseGrid(Ic,Jc)) / 2.0;
     fineGrid(If-1,Jf-1) = (coarseGrid(Ic-1,Jc-1) + coarseGrid(Ic-1,Jc) +
                            coarseGrid(Ic,Jc-1) + coarseGrid(Ic,Jc)) / 4.0;
}
```

Fig. 6. Example: High-Level user code in function interpolate2D using classes floatArray and Range

```
void interpolate2D(class floatArray &fineGrid,class floatArray &coarseGrid)
{
  int _var_9; int _var_8; int _var_7; int _var_6; int _var_5;
  int _var_4; int _var_3; int _var_2; int _var_1; int _var_0;
  int fineGridSizeX = (fineGrid.length(0));
  int fineGridSizeY = (fineGrid.length(1));
  int coarseGridSizeX = (coarseGrid.length(0));
  int coarseGridSizeY = (coarseGrid.length(1));
  // Interior fine grid points
  class Range If(2,_var_2 = fineGridSizeX - 2,2);
  class Range Jf(2,_var_3 = fineGridSizeY - 2,2);
  class Range Ic(1,coarseGridSizeX,1); class Range Jc(1,coarseGridSizeY - 1,1);
  for (_var_1 = 0; _var_1 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_1 += 1) {
    for (_var_0 = 0; _var_0 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_0 += 1) {
      fineGrid.elem(_var_0 * 2 + 2,_var_1 * 2 + 2)
        = coarseGrid.elem(_var_0 * 1 + 1,_var_1 * 1 + 1);
    }
  }
  for (_var_5 = 0; _var_5 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_5 += 1) {
    for (_var_4 = 0; _var_4 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_4 += 1) {
      fineGrid.elem(2 + -1 * 1 + _var_4 * 2,_var_5 * 2 + 2)
        = (coarseGrid.elem(1 + -1 * 1 + _var_4 * 1,_var_5 * 1 + 1)
        + coarseGrid.elem(_var_4 * 1 + 1,_var_5 * 1 + 1)) / 2.0;
    }
  }
  for (_var_7 = 0; _var_7 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_7 += 1) {
    for (_var_6 = 0; _var_6 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_6 += 1) {
      fineGrid.elem(_var_6 * 2 + 2,2 + -1 * 1 + _var_7 * 2)
        = (coarseGrid.elem(_var_6 * 1 + 1,1 + -1 * 1 + _var_7 * 1)
        + coarseGrid.elem(_var_6 * 1 + 1,_var_7 * 1 + 1)) / 2.0;
    }
  }
  for (_var_9 = 0; _var_9 <= -1 + (_var_3 + -1 * 2 + 1) / 2; _var_9 += 1) {
    for (_var_8 = 0; _var_8 <= -1 + (_var_2 + -1 * 2 + 1) / 2; _var_8 += 1) {
      fineGrid.elem(2 + -1 * 1 + _var_8 * 2,2 + -1 * 1 + _var_9 * 2)
        = (coarseGrid.elem(1 + -1 * 1 + _var_8 * 1,1 + -1 * 1 + _var_9 * 1)
        + coarseGrid.elem(1 + -1 * 1 + _var_8 * 1,_var_9 * 1 + 1)
        + coarseGrid.elem(_var_8 * 1 + 1,1 + -1 * 1 + _var_9 * 1)
        + coarseGrid.elem(_var_8 * 1 + 1,_var_9 * 1 + 1)) / 4.0;
    }
  }
}
```

Fig. 7. Example: Function interpolate2D with translated array operations only

```
void interpolate2D(class floatArray &fineGrid,class floatArray &coarseGrid)
{
  int coarseGrid_length_2 = (coarseGrid.Array_Descriptor.Array_Domain.getLength(1));
  int coarseGrid_length_1 = (coarseGrid.Array_Descriptor.Array_Domain.getLength(0));
  int coarseGrid_stride_2 = (coarseGrid.Array_Descriptor.Array_Domain.Stride[1]);
  int coarseGrid_stride_1 = (coarseGrid.Array_Descriptor.Array_Domain.Stride[0]);
  int coarseGrid_size_2 = (coarseGrid.Array_Descriptor.Array_Domain.Size[1]);
  int coarseGrid_size_1 = (coarseGrid.Array_Descriptor.Array_Domain.Size[0]);
  float *coarseGrid_pointer = (coarseGrid.getDataPointer());
  int fineGrid_length_2 = (fineGrid.Array_Descriptor.Array_Domain.getLength(1));
  int fineGrid_length_1 = (fineGrid.Array_Descriptor.Array_Domain.getLength(0));
  int fineGrid_stride_2 = (fineGrid.Array_Descriptor.Array_Domain.Stride[1]);
  int fineGrid_stride_1 = (fineGrid.Array_Descriptor.Array_Domain.Stride[0]);
  int fineGrid_size_2 = (fineGrid.Array_Descriptor.Array_Domain.Size[1]);
  int fineGrid_size_1 = (fineGrid.Array_Descriptor.Array_Domain.Size[0]);
  float *fineGrid_pointer = (fineGrid.getDataPointer());
  int _var_9; int _var_8; int _var_7; int _var_6; int _var_5;
  int _var_4; int _var_3; int _var_2; int _var_1; int _var_0;
  int fineGridSizeX = (fineGrid_length_1);
  int fineGridSizeY = (fineGrid_length_2);
  int coarseGridSizeX = (coarseGrid_length_1);
  int coarseGridSizeY = (coarseGrid_length_2);

  // Interior fine grid points
  class Range If(2,_var_2 = fineGridSizeX - 2,2);
  class Range Jf(2,_var_3 = fineGridSizeY - 2,2);
  class Range Ic(1,coarseGridSizeX,1);
  class Range Jc(1,coarseGridSizeY - 1,1);

  for (_var_1 = 0; _var_1 <= -1 + (_var_3 + -1) / 2; _var_1 += 1) {
    for (_var_0 = 0; _var_0 <= -1 + (_var_2 + -1) / 2; _var_0 += 1) {
      fineGrid_pointer[_var_0 * 2 + 2 + (_var_1 * 2 + 2)
                      * fineGrid_stride_1 * fineGrid_size_1]
        = coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1];
      fineGrid_pointer[2 + -1 * 1 + _var_0 * 2 + (_var_1 * 2 + 2)
                      * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (_var_1 * 1 + 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1]) / 2.0;
      fineGrid_pointer[_var_0 * 2 + 2 + (2 + -1 * 1 + _var_1 * 2)
                      * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[_var_0 * 1 + 1 + (1 + -1 * 1 + _var_1 * 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1]) / 2.0;
      fineGrid_pointer[2 + -1 * 1 + _var_0 * 2 + (2 + -1 * 1 + _var_1 * 2)
                      * fineGrid_stride_1 * fineGrid_size_1]
        = (coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (1 + -1 * 1 + _var_1 * 1)
                             * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[1 + -1 * 1 + _var_0 * 1 + (_var_1 * 1 + 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (1 + -1 * 1 + _var_1 * 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1]
        + coarseGrid_pointer[_var_0 * 1 + 1 + (_var_1 * 1 + 1)
                            * coarseGrid_stride_1 * coarseGrid_size_1]) / 4.0;
    }
  }
}
```

Fig. 8. Example: optimized function interpolate2d (translating array operations + loop fusion)

have been applied only to expressions composed of built-in types. We enable the optimization of statements and expressions of high-level user-defined abstractions. Additional annotations are used to specify the semantics of user-defined types which are not determined by the existing program analysis. The additional semantics are utilized in optimizing user-defined abstractions.

As an example, we have implemented an adapted constant-propagation/folding algorithm to automatically infer the symbolic properties of arbitrary user-defined objects. In Figure 5, two annotations, *has-value* and *restrict-value*, are used to describe the properties. Specifically, *has-value* declares that class *floatArray* has two properties: the array dimension and the length of each dimension i, and that class *Range* has three properties, *base*, *len* and *stride*, for selecting subsets of elements from arrays. Similarly, the annotation *restrict-value* declares how properties of user-defined types can be implied from function calls. For example, if "*floatArray::operator()(int index)*" is used to access the element of an *floatArray* object arr, we know that arr must have a single dimension, and it will remain single-dimensional until some other operator modifies its shape. We have also combined the symbolic property analysis with loop optimizations to automatically determine the shapes of user-defined abstractions [8].

Using these annotations as additional semantic information we can translate high-level array operations to a sequence of loops and perform loop fusion. For example, in Fig. 6 we show the high-level code of a 2D interpolation. The essential information from annotations is the element-wise modification of the array (modify annotation). The translated source code is shown in Fig. 7. Additionally, we applied loop fusion to this version and obtained an even more efficient version of the program, shown in Fig. 8. Both optimization steps use information from the annotations. In the next section we discuss the performance of these optimized versions and other benchmarks.

5 Experimental Results

This section presents some preliminary results from applying loop optimizations to several kernels written using the A++/P++ Library [1], an array class library that supports both serial and parallel array abstractions with a single interface. We selected our kernels from the Multigrid algorithm for solving elliptic partial differential equations. The Multigrid algorithm consists of three phases: relaxation, restriction, and interpolation, from which we selected both interpolation and relaxation on one, two, and three dimensional problems.

Our experiments aim to validate two conclusions: our approach can significantly improve the performance of numerical applications, and our approach is general enough for optimizing a large class of applications using object-oriented abstractions. The kernels we used, though small, use a real-world array abstraction library and are representative of a much broader class of numerical computations expressed using sequences of array operations. All six kernels (one, two and three-dimensional interpolation and relaxation) benefited significantly from our optimizations.

(a) Interpolation results

	Interp1D				Interp2D				Interp3D			
array size	orig (sec)	translate only	translate + fusion	fusion only	orig (sec)	translate only	translate + fusion	fusion only	orig (sec)	translate only	translate + fusion	fusion only
50	4.833	1.915	2.131	1.113	7.000	3.034	3.932	1.296	9.166	2.497	3.184	1.275
75	5.000	4.142	4.519	1.091	7.000	2.766	3.131	1.132	9.333	3.021	3.813	1.262
100	5.333	2.593	2.899	1.118	7.000	2.753	3.247	1.179	9.333	2.929	3.767	1.286
125	7.666	2.853	4.228	1.482	9.833	3.304	3.882	1.175	10.666	3.214	4.442	1.382
150	9.166	2.390	4.214	1.763	11.166	2.897	4.542	1.568	12.333	2.871	4.189	1.459
175	11.366	2.630	4.618	1.756	12.833	2.893	4.964	1.716	15.766	3.403	5.264	1.547
200	11.000	2.419	4.289	1.773	14.799	3.161	5.348	1.692	13.799	2.514	4.211	1.675

(b) Red-black relaxation results

	RedBlack1D				RedBlack2D				RedBlack3D			
array size	orig (sec)	translate only	translate + fusion	fusion only	orig (sec)	translate only	translate + fusion	fusion only	orig (sec)	translate only	translate + fusion	fusion only
50	11.500	2.178	5.338	2.451	17.166	1.650	3.344	2.026	22.499	3.260	3.445	1.057
75	14.999	1.728	6.692	3.872	16.666	1.627	3.280	2.016	27.332	3.938	3.776	0.959
100	26.166	3.540	11.852	3.348	32.165	2.672	5.146	1.926	35.665	4.744	4.176	0.880
125	32.499	1.960	12.327	6.289	41.498	2.418	4.421	1.828	45.998	4.685	3.895	0.831
150	35.165	2.865	13.885	4.847	46.665	2.134	4.643	2.176	53.498	5.272	4.440	0.842
175	38.132	2.344	15.270	6.513	52.065	2.514	5.378	2.140	64.531	6.238	5.701	0.914
200	38.598	3.125	15.117	4.838	53.398	2.501	6.117	2.446	67.797	6.703	5.384	0.803

Table 1. Performance results (*orig*: elapsed time of original versions written using array abstractions — different numbers of iterations were run for different problem sizes; *translate-only*: speedups from translating array abstractions into low-level C implementations; *translate+fusion*: speedups from both array translation and loop fusion; *fusion-only*: speedups from applying loop fusion alone.)

We generated three versions for each kernel: the original version (*orig*) using array abstractions, the *translate-only* version auto-optimized by translating array operations into low level C implementations, and the *translate+fusion* version auto-optimized both with array translation and loop fusion. For the two-dimensional interpolation code Fig. 6 shows the *original*, Fig. 7 the *translate-only* version, and Fig. 8 the *translate+fusion* version.

The original versions of all kernels each have 20-60 lines of code, (they look simple because they are written using array abstractions). After translating array operations into explicit loops, each kernel contains 2-8 loop nests which are then considered for loop optimization. Each loop nest has 1-3 dimensions, depending on the dimensionality of the arrays being modified.

We measured all versions on a Compaq AlphaServer DS20E. Each node has 4GB memory and two 667MHz processors. Each processor has L1 instruction and data caches of 64KB each, and 8MB L2 cache. We used the Compaq vendor C++ compiler with the highest level of optimization, and measured the elapsed-time of each execution. Table 1 present our measurements using multiple array sizes.

From Table 1(a), in nearly all cases the translation of the array abstractions results in significant improvements. But applying loop fusion improves the performance further by 20%-75%. This validates our belief that loop optimization is a significant step further toward fully recovering the performance penalty of using high-level array abstractions.

From Table 1(b), the dominant performance improvements come from translating array abstractions into low-level implementations(*translate-only*). Loop fusion can further improve performance by 2.3-6.5 times for one and two-dimensional relaxation kernels, but for three-dimensional relaxation, it showed only slight improvement (5%) for small arrays(50) and degraded performance (up to 20%) for large arrays. Here the performance degradation is due to increased register pressures from the much larger fused loop bodies in the three-dimensional case. We are working on better algorithms to selectively apply loop fusion.

The final codes generated by our optimizer are very similar to the corresponding C programs that programmers would manually write. Consequently, we believe that their performance would also be similar. Further, because programmers usually don't go out-of-the-way in applying loop optimizations, our techniques can sometimes perform better than hand-written code. This is especially true for the red-black relaxation kernels, where the original loops need to be re-aligned before fusion and a later loop-splitting step is necessary to remove conditionals inside the fused loop nests. Such complex transformations are much more easily applied automatically by compilers than manually by programmers.

6 Related Work

Related work on the optimization of libraries in telescoping languages [9] shares similar goals as our research. The SUIF compiler [10] and OpenC++ [11] each provided a programmable level of control over the compilation of applications in support of optimizing user-defined abstractions. The Broadway compiler [12] uses general *annotation languages* to guide source code optimizations. Within ROSE, we provide both an open compiler infrastructure for programmers to define their own optimizations and a collection of annotation mechanisms for programmers to exploit predefined traditional compiler optimizations. Template Meta-Programming[13, 14] has also been used to optimize user-defined abstractions, but is effective only when optimizations are isolated within a single statement. Optimizations across statements, such as loop fusion, is beyond the capabilities of template meta-programming.

A rich set of compiler optimization techniques have been developed to improve the performance of applications, including a collection of loop transformations. These transformations by default can only optimize operations on primitive types, whose semantics are known by the compilers. To extend these optimizations to user-defined abstractions, Wu, Midkiff, Moreira and Gupta [15] proposed *semantic inlining*, which treats specific user-defined types as primitive types in Java. Artigas, Gupta, Midkiff and Moreira [16] devised an *alias versioning* transformation that creates alias-free regions in Java programs so that loop optimizations can be applied to Java primitive arrays and the array abstractions from their library. Wu and Padua [17] investigated automatically parallelization of loops operating on user-defined containers, but assumed that their compiler knew about the semantics of all operators. All the above approaches apply compiler techniques to optimize library abstractions. However, by encoding the knowledge within their compilers, these specialized compilers cannot be used to optimize abstractions in general other than those in their libraries. In contrast, we target optimizing general user-defined abstractions by allowing programmers to classify their abstractions and to explicitly communicate semantics information with the compiler.

7 Conclusions

User-defined abstractions are productive in the development of application codes, but the abstraction penalty is often not acceptable for scientific computing. We have presented an approach that reduces this penalty such that the performance of user-defined abstractions becomes acceptable for high-performance computing, allowing to use these abstractions to achieve higher productivity in the development of scientific applications.

We have demonstrated that leveraging semantics of user-defined abstractions can provide significant opportunities for our optimizations and identified an annotation approach to specify relevant user-defined semantics. Using these annotations, we built an automated transformation approach greatly simplifying the otherwise explicit specification of program transformations using more traditional approaches (such as the other mechanisms in ROSE).

We expect that additional research work on the classification of general abstractions will lead to a more useful and practical optimization approach tailored to the domain specific optimization opportunities of user-defined abstractions. The success of such an approach depends upon other research areas, including the verification of transformations, verification of annotations, and the semantic classification of general user-defined abstractions.

References

1. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
2. Daniel Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16, Issue 2-3:293–302, February 2004.
3. Markus Schordan and Daniel Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, August 2003.
4. Edison Design Group. http://www.edg.com.
5. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

6. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
7. Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG: fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, April 1992.
8. Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.
9. Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.
10. M. S. Lam S. P. Amarasinghe, J. M. Anderson and C. W. Tseng. The suif compiler for scalable parallel machines. In *in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.
11. Shigeru Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.
12. Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, January 2000.
13. Todd Veldhuizen. Expression templates. In S.B. Lippmann, editor, *C++ Gems*. Prentice-Hall, 1996.
14. Federico Bassetti, Kei Davis, and Dan Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In Ishikawa et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97*, volume 1343 of *LNCS*. Springer, 1997.
15. Peng Wu, Samuel P. Midkiff, Jose E. Moreira, and Manish Gupta. Improving Java performance through semantic inlining. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 1999.
16. Pedro V. Artigas, Manish Gupta, Samuel Midkiff, and Jose Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.
17. Peng Wu and David Padua. Containers on the parallelization of general-purpose Java programs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct 1999.