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 classified to support their automated optimization and by leveraging these additional semantics we enable the library specific optimization of application codes. We believe that such an approach entails the use of formal methods.
We describe ROSE, a framework for building source-to-source translators, used for the high-level optimization of scientific applications. 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. ROSE allows the recognition of such abstractions and the optimization of their use in applications. We show how ROSE can be used to utilize the additional semantics within the compile-time optimization and present promising results.

1 Introduction

User-defined high-level abstractions are productive in the development of application codes. Unfortunately the use of high-level abstractions usually introduces a penalty in performance due to indirection, insufficient global optimizations of compilers, or the lack of program analysis to infer high-level semantics and properties of user-defined abstractions within acceptable time bounds. Such lack of information about semantics of user-defined abstractions in libraries disallows to perform optimizations of application codes using the library's provided abstractions.

If an optimization is known for a user-defined abstraction but the required properties of this abstraction for guaranteeing the correctness of the optimization cannot be established, we face a (known) performance penalty because the more efficient version of the code is not generated. Such a transformation can be carried out manually but this clearly reduces productivity because it would also require to maintain the manually optimized code. Our approach aims at fully automating such optimizations.

We bridge the gap of unknown high-level semantics by providing additional information through annotations. This additional information is used to enable optimizations specific to user-defined abstractions. We present a compile-time approach for the optimization of scientific applications using the semantics of library abstractions and demonstrate the approach toward the definition of annotations for abstractions using a specific abstraction. The abstraction chosen for optimization in our example is an array abstraction contained within an array class library which is used in our scientific applications.

Given appropriate annotations and transformations, the mechanisms we present to optimize high-level abstractions are sufficient to optimize arbitrary abstractions. Thus we have addressed numerous aspects of the more general problem within the current work. Many aspects of future work will depend upon either manual or automated generation of annotations, to classify the semantic properties of general abstractions. We describe the concrete work that has been done to develop an annotation based approach to guide the automated optimization of high-level abstractions, starting with a motivating example.

1.1 Motivating Example

Figure 1 shows an example of a high-level abstraction. It is an array abstraction which is used within our current scientific applications and therefore forms an attractive example problem.

The class Range allows to define integer ranges with a fixed stride for iterating, and a class doubleArray where Range objects serve as parameters to several functions for iterating on 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. The user-defined types Range and doubleArray allow to define such a computation without the use of any loop constructs in the application code. Using such classes, the user is freed from the burden of writing loops for each dimension of an array. In the example we use a two-dimensional array with type double as element type.

The function "*main*" consists of four lines. In line 2 two arrays A and B, each of size 100x100, are created. In line 3 Range objects, defining ranges from 1 to 98 with stride 1 are created. Index 0 and 100 are not used in the example. 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 doubleArray. In this simple example we have restricted the available functions to "*+*" and "*sin*". The full class consists of about 80 different operators.

For our optimizations it is relevant how the iterations are performed on the array, whether the array points are read-only or modified, and whether aliases are used to express sharing of data structures. For optimizations on the lowered code it is relevant which iterations can be combined into fused loops. We shall show how such properties can be specified using our annotation language in section 4.

```
class Range {
public:
    Range ( int base, int bound, int stride );
    Range operator+ ( int i );
    Range operator- ( int i );
};
class doubleArray {
public:
    doubleArray ( int i, int j );
    doubleArray & operator= ( const doubleArray & X );
    friend doubleArray operator+ ( const doubleArray & X, const doubleArray & Y );
    doubleArray operator() ( const Range & I, const Range & J );
    friend doubleArray & sin ( const doubleArray & X );
};
int main () {                                           /* 1 */
    doubleArray 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: Snippet of header file and program to be optimized.

In the remaining sections we present the ROSE[1, 2] architecture used for implementing the presented approach, the annotation based mechanism for the optimization of array abstractions, an example problem 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) [3] 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,

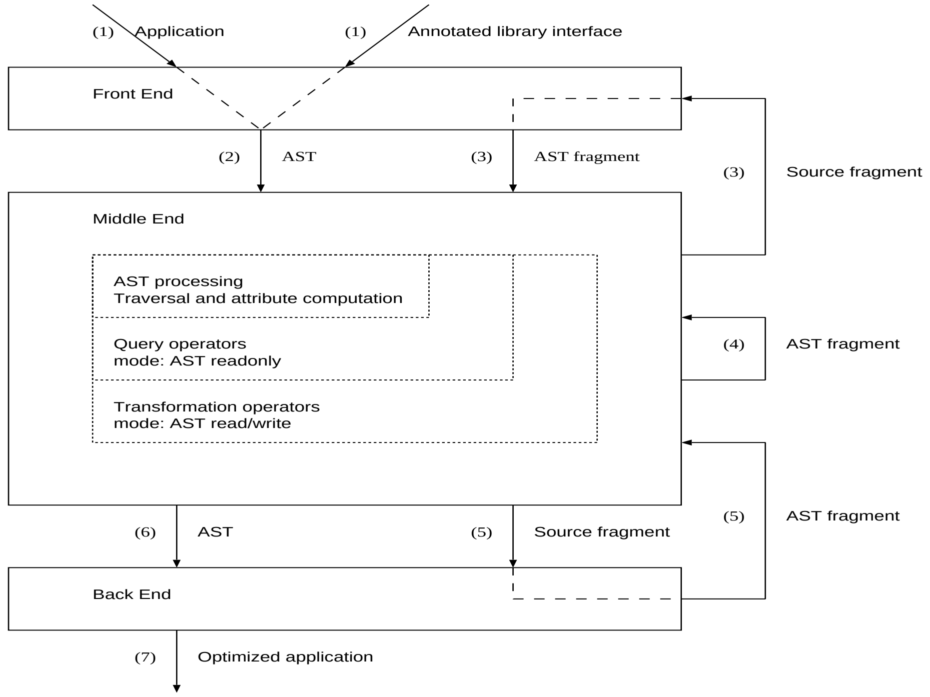

Fig. 2. ROSE Source-To-Source architecture

based on Sage II and Sage++[4]. 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).

2.2 Mid End

The mid end allows 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 allows computation of 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 [5] and Frankie Erse's C/C++ port, can be found in [2].

An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. Transformation operators can be built by using the AST processing mechanism in combination with AST restructuring operations. In Fig. 2 steps 3,4,5 show how the ROSE architecture also allows using 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. 3, 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 allow 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. This allows building 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

The annotations of the AST consist of type information obtained from the EDG front end and user-defined attributes which allow attaching results of attribute computations to AST nodes. These results can be accessed by subsequent AST processing steps and allow the composition of different AST operators. Annotations can also specify additional semantic information. This 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, a separate annotation file.

3 Transformation Operators

A transformation operator consists of a pre-condition to hold (based on the AST annotations which are computed in the analysis phase) and a restructuring sequence which can be applied safely if the pre-condition holds. A restructuring sequence consists of fragment operations which we shall discuss in detail.

An optimization requires an analysis of the program 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 computing attributes and fixed point algorithms for flow sensitive analysis can be applied on the control flow graph. The analysis results are attached to the AST as annotations.

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 [2]. Bottom Up Rewrite Systems (BURS), such as burg [6], 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.

The correctness of a transformation is addressed by ensuring that the sequence of the restructuring operations on the AST preserves the semantics of the program. A transformation operator consists of a pre-condition to hold (based on the AST annotations which are computed in the analysis phase) and a sequence of restructuring operations which can be applied safely if the pre-condition holds. 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 allows performing a basic restructuring operation such as insert, delete, or replace AST fragments. The target location in the AST can be absolute or relative. The fragment to be inserted can be specified as source fragment or AST fragment. 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. A source fragment s 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 l_{abs}. From the syntactic context the prefix, $s_\triangleleft$, is computed such that all declarations, opening scopes, and function signatures 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.). Hence, a source fragment, $s_\Box$, is valid if $frontend(s_\triangleleft + s_\Box + s_\triangleright)$ succeeds, i.e. all syntactic and semantic checks succeed and a corresponding AST fragment, $ast_\Box$, can be generated.

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. 3.1 an overview of the most important fragment operators is given. The fragment operators allow rewriting the AST by specifying absolute or relative target locations. A relative location l_{rel} allows specification of a target location in an AST relative to an absolute location l_{abs}. The operator *location* can map a relative location l_{rel} with respect to an absolute location l_{abs} and a given AST containing the absolute location l_{abs}, to another absolute location in the same AST according to L_{rel}. 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 the target location as the statement outside the loop-scope right before the loop. We have defined several classifications of such relative target locations which were 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. 3.1, *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 defined.

4 Predefined Optimizations

A large set of compiler optimizations, including both reordering transformations such as loop fusion/fission and blocking, and instruction 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 disabled 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(a). In the following, we use the annotation examples in Figure 4(b) to further illustrate the techniques.

4.1 Enabling Transformations

The most significant enabling transformations for library abstractions is inlining, which eliminates function calls by merging the implementations of the functions within their calling contexts. Suppose the compiler has access to all the source code of a library, theoretically, inlining the library code could permit all necessary program analysis and thus allow the compiler to discover/unlock 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, 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.

```
<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>
```

(a) grammar

```
class doubleArray:
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$x:0,dim-1=this.getLength(x); }
operator doubleArray::operatpr =
(const doubleArray& that):
modify {this}; read {that}; alias none;
modify-array (this) {
      dim = that.dim; len(i) = that.len(i);
      elem(i$x:1:dim) = that.elem(i$x); };
operator +(const doubleArray& a1,double a2):
modify none; read{a1,a2}; alias none;
new-array () { dim = a1.dim; len(i) = a1.len(i);
             elem(i$x:1:dim) = a1.elem(i$x)+a2; };
operator doubleArray::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); };
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 doubleArray::operator() (int index) :
inline { this.elem(index) };
restrict-value { this = { dim = 1; };};
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; };};
```

(b)example

Fig. 4. Annotation language

In our annotation language, the programmers can not only direct 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. It is used in Figure 4(b) for function "*doubleArray::operator()(int)*", which is declared as a subscripted access of the current *doubleArray* 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.

In Figure 4(b), the *is-array* annotation declares that the class *doubleArray* 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}. Similarly, the operator "*doubleArray::operator= (const doubleArray& that)*" is declared to have *modify-array* semantics; that is, it performs element-wise modification of the current array. The operator "+*(const doubleArray&* 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 "*doubleArray::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. For details in using the annotations for loop optimizations, see [7].

4.3 Instruction Level Transformations

To generate efficient code, most compilers include instruction level optimizations that eliminate redundant computations or replace expensive computations with cheaper ones. The most commonly used optimizations include constant propagation, constant folding, strength reduction, redundant expression elimination, and dead code elimination. Most of these optimizations have been applied only to integer expressions that contain no obscure function calls, and can be extended with annotation interfaces for optimizing high-level user-defined abstractions.

As example, we have implemented an adapted constant-propagation/folding algorithm to automatically determine the symbolic value properties of arbitrary user-defined objects. In Figure 4, two annotations, *has-value* and *restrict-value*, are used to describe the properties. Specifically, *has-value* declares that class *doubleArray* 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 "*doubleArray::operator()(int index)*" is used to access the element of an *doubleArray* 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 combined the symbolic property analysis with loop optimizations to automatically determine the shapes of user-defined Fortran-array abstractions. For more detail, See [7].

5 Experimental Results

This section presents some preliminary results from applying loop optimizations to several kernels written using the A++/P++ Library [8], 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(specifically, red-black 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.

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. As example, Figure 5 shows the *original*, *translate-only* and *translate+fusion* versions for the one-dimensional interpolation code, the simplest of all kernels. 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.

```
void interpolate1D (floatArray& fineGrid, floatArray& coarseGrid) {
  int fineGridSize = fineGrid.getLength(0), coarseGridSize = coarseGrid.getLength(0);
  Range If (2,fineGridSize-2,2), Ic (1,coarseGridSize-1,1);
  fineGrid(If) = coarseGrid(Ic);
  fineGrid(If-1) = (coarseGrid(Ic-1) + coarseGrid(Ic)) / 2.0;
}
```

(a) original version

```
void interpolate1D (floatArray& fineGrid, floatArray& coarseGrid ) {
  int fineGridSize = fineGrid.getLength(0), coarseGridSize = coarseGrid.getLength(0);
  Range If (2,fineGridSize-2,2), Ic (1,coarseGridSize-1,1);
  float* fineGridPointer = findGrid.getPointer(), *coarseGridPointer = coarseGrid.getPointer();
  for (int _i = 0; _i < (fineGridSize - 3) / 2; _i += 1)
    fineGridPointer[_i * 2 + 2] = coarseGridPointer[_i + 1];
  for (int _j = 0; _j < (fineGridSize - 3) / 2; _j += 1)
    fineGridPointer[_j * 2 + 1] = coarseGrid[_j] + coarseGridPointer[_j + 1] / 2.0;
}
```

(b) translating array operations only

```
void interpolate1D (floatArray& fineGrid, floatArray& coarseGrid ) {
  int fineGridSize = fineGrid.getLength(0), coarseGridSize = coarseGrid.getLength(0);
  Range If (2,fineGridSize-2,2), Ic (1,coarseGridSize-1,1);
  float* fineGridPointer = findGrid.getPointer(), *coarseGridPointer = coarseGrid.getPointer();
  for (int _i = 0; _i < (fineGridSize - 3) / 2; _i += 1)
    fineGridPointer[_i * 2 + 2] = coarseGridPoiner[_i + 1];
    fineGridPointer[_j * 2 + 1] = coarseGridPointer[_j] + coarseGridPointer[_j + 1] / 2.0;
}
```

(c) translating array operations + loop fusion

Fig. 5. Example: 1D interpolation

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 dominate 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

(a) Interpolation results

	Interp1D				Interp2D				Interp3D			
array size	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te 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)	transla te only	translate + fusion	fusion only	orig (sec)	transla te only	translate + fusion	fusion only	orig (sec)	transla te 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.)

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 allows to reduce 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). The evaluation was performed using an array abstraction. 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.

References

1. 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.
2. 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.
3. Edison Design Group. http://www.edg.com.

4. 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.
5. Hanspeter Moessenboeck. Coco/R - A generator for production quality compilers. In *LNCS477, Springer*, 1991.
6. 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.
7. Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. Technical Report UCRL-CONF-202762, Lawrence Livermore National Laboratory, Livermore, CA, 2004.
8. 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.
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.