UCRL-PROC-217573

Annotating user-defined abstractions for optimization

D. Quinlan, M. Schordan, R. Vuduc, Q. Yi

December 7, 2005

IPDPS Workshop on Performance Optimization of High-Level Languages and Libraries
Rhodes, Greece
April 25, 2006 through April 29, 2006

Annotating User-Defined Abstractions for Optimization

Dan Quinlan[*], Markus Schordan[†], Richard Vuduc[*], Qing Yi[‡]
[*]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, USA.
{DQUINLAN,RICHIE}@LLNL.GOV
[†]Institute of Computer Languages, Vienna University of Technology, Austria.
MARKUS@COMPLANG.TUWIEN.AC.AT
Department of Computer Science, University of Texas at San Antonio, USA.
QINGYI@CS.UTSA.EDU

I. Introduction

We consider the problem of improving the performance of scientific computing applications that rely on user-defined high-level abstractions to manage software and hardware complexity. Although conventional compilers implement a wide range of optimization techniques, they frequently miss opportunities to optimize the use of abstractions, largely because they are not designed to recognize and use the relevant semantic information about such abstractions. As a result, developers today either accept the idea of an "abstraction penalty" and live with relatively poor performance, or manually rewrite their code to use lower-level constructs and obtain better performance at the cost of maintainability and portability.

In this paper, we propose a set of annotations to help communicate high-level semantic information about abstractions to the compiler, thereby enabling the large body of traditional compiler optimizations to be applied to the use of those abstractions. These annotations explicitly describe properties of the abstraction needed to guarantee the applicability and profitability of particular optimizing transformations.

Our annotations are influenced by our practical experience with optimizations that have or are likely to have a significant impact on applications used throughout the U.S. Department of Energy (DOE) research laboratories. Though not all-inclusive, these annotations permit a broad variety of classical optimizations, including memoization, reordering, data layout transformations, and inlining and specialization, to be applied to the use of abstractions. This paper generalizes our earlier research on the optimization of array abstractions [22] to arbitrary user-defined abstractions.

Figure 1 shows the simplified example of a user-defined abstraction that exhibits many features of typical unstructured mesh computations within scientific computing applications. Specifically, the *Mesh* class represents a user-defined abstraction, and the *compute()* procedure is a user's algorithm that operates on a *Mesh* object. The *Mesh* class is conceptually a container of nodes and edges, where an edge connects two nodes, and *Mesh* provides a means to iterate over either nodes or edges. An important property of *Mesh* is that there is a many-to-one and onto mapping (surjection) of edges to nodes.

Figure 1. **User-defined abstraction example.**

```
class Node { // ...
   public: int id(); double eval(double a);
};
class Edge { // ...
   public: Node *node1(); Node *node2();
};
class Mesh { // ...
  public:
    Edge* get_edge(int i);
    Node* get_node(int i);
    int node_size(); int edge_size();
};

void compute(Mesh& m, double a) {
  for (int i = 0; i < m.edge_size(); ++i) {
      Edge* e = m.get_edge(i);
      // ...
      double a = e->node1()->eval(a);
      double b = e->node2()->eval(a);
      bar (a, b);
      // ...
    }
}
```

The procedure *compute()* implements some algorithm that is most naturally expressed as iteration over edges, but repeatedly calls an expensive and side-effect free function *eval* on each node. Because of the surjection, we can optimize *compute()* by precomputing *eval* on all nodes (memoization), yielding the optimized implementation shown in Figure 2. White, *et al.*, showed a 2× speedup by applying this specific memoization to a realistic benchmark extracted from a DOE code [19].

Figure 3 shows how we specify the properties of *Mesh* as annotations, to enable the translation of Figure 1 into Figure 2. The annotations describe aliasing properties of the data abstractions, and side-effect properties of the function abstraction *eval()*. Moreover, we can specify properties of the data structure (such as lower and upper bounds on the ratio of edges to nodes) that could influence subsequent profitability analyses.

The optimized code in Figure 2 is more complex and

Figure 2. ***compute()* optimized by memoization.**

```
void compute_optimized(Mesh& m, double a) {
   vector<double> a_precomp (m.node_size ());
   for ( int  i = 0;  i < m.node_size();  ++i) {
        Node *n = m.get_node(i);
        a_precomp[n->id()] = n->eval(a);
   }

   for ( int  i = 0;  i < m.edge_size();  ++i) {
        Edge *e = m.get_edge(i);
        // ...
        double  a  = a_precomp[e->node1()->id()];
        double  b  = a_precomp[e->node2()->id()];
        bar  (a,  b);
        // ...
     }
}
```

Figure 3. **Annotations for *Mesh*.**

```
class  Node :  has_value  {  id  =  this . id ();  }
class  Edge :  has_value  {
   n1 =  this .node1();  n2 =  this .node2();
   };
operator  Node::eval(double  a)  :
   read  {this ,a};  modify none;  alias  none;
class  Mesh :  has_value{
     nsize  =  this . node_size ();   esize  =  this . edge_size ();
     nodes(i :0: nsize)=  this . get_node(i );
     edges(i :0: esize )  =  this . get_edge(i );
     };
   restrict_value   { nodes(i ). id  ≠nodes(j ). id ;  }
  never_alias (edges(i ). n1)  = edges(i ). n2;
  never_alias (edges(i ))  =   edges(j )  :  j  ≠i  ;
  never_alias (nodes(i ))  =   nodes(j )  :  j  ≠i  ;
  must_alias (nodes(j ))  =   edges(i ). n1 or  edges(i ). n2;
   restrict_value   {esize  ≥nsize  ∗ k1; esize ≤nsize ∗ k2}
```

difficult to debug, so it is undesirable to introduce by hand everywhere. Additionally, whether the optimization is profitable will vary by computer architecture. Although the optimization can be automated as a source-to-source transformation, the transformation is heavily dependent on the semantics of the abstraction so that a conventional compiler is unlikely to discover such an optimization opportunity. Vendor compilers rarely introduce optimizations that require additional storage, due to the complexity of their analysis. In the following sections, we use this example to motivate the need for annotations to support the difficult program analysis required to know when such optimizations may be used profitably.

Our annotations, presented in Section III, provide an open interface for developers to communicate the semantics of their abstractions to the compiler. These annotations complement program analysis when compiler analysis is insufficient to enable optimizations, as discussed in Section II. Whether an abstraction is annotated automatically or specified by the developer, traditional optimizations can be naturally extended to use the annotations and then applied to uses of the abstraction. We describe such extensions in Section IV, using examples of optimizations important to DOE laboratory applications. We are implementing these ideas in the ROSE source-to-source compiler infrastructure (Section V) [14], [16].

II. Optimizing User-Defined Abstractions with Traditional Optimizations

In contrast to built-in types, user-defined abstractions are constructed using the base language. If a compiler aims at optimizing the use of these abstractions, it must first infer the abstraction semantics through analysis of their implementations. As a compiler must preserve the input program's semantics, it performs a conservative analysis of the program and compromises precision with respect to correctness. Consequently, we might lose opportunities for optimization. Therefore the compiler often performs very limited optimizations when some properties of the user-defined abstractions cannot be established automatically by the analysis. Most of the missing optimizations can be applied by a compiler if the missing information from analysis is provided. We give a short overview of traditional optimizations that can be applied to code that uses abstractions, and summarize their salient properties. The optimizations can be separated into the following categories.

- *Memoization optimizations*. These optimizations use dataflow analysis to determine results of computations that can be saved (memoized) for later reuse, thereby avoiding redundant computation. For example, to perform common subexpression elimination the analysis must compute *available expressions* (expressions whose results have already been computed) at the entry of each basic block. If a statement contains a function call for which program analysis cannot precisely determine possible side effects, the function call may potentially render the previous computed results of all expressions invalid. Therefore, those expressions must be considered not available and must be recomputed after the function call. Similar problems exist for loop invariant code motion (which uses loop invariant detection analysis) and strength reduction optimization (which uses induction variable detection analysis). A special case of memoization is *dead code elimination*, where computations that will never be used are eliminated completely. If properties such as side-effects of methods are specified using annotations, these memoization optimizations can be applied to code fragments that use abstractions unknown to the compiler. We discuss this in more detail below.
- *Reordering optimizations*. These optimizations perform dependence analysis to determine reordering constraints between each pair of statement (or instruction) instances. Statements (or instructions) are then reordered to better utilize the computer resources. Examples of reordering optimizations include instruction scheduling for better CPU utilization, loop transformations such as loop fusion and blocking for better memory hierarchy performance, and automatic parallelization and communication optimizations for better utilization of multiprocessors.

- *Data layout optimizations*. These optimizations aim at rearranging the layout of data structures to accommodate resource constraints of computers. Examples include register allocation, scalar replacement, and array padding. Our annotations permit specifying several properties of arrays.
- *Abstraction inlining and specialization*. Such optimizations aim at eliminating or reducing the overhead of introducing user-defined abstractions in programs. Such overheads include the cost of making function calls, missed optimization opportunities due to abstraction boundaries, as well as inefficient data grouping and additional indirections due to the necessity of data abstraction. Optimizations of this category include procedure inlining and specialization, data structure splitting, and elimination of indirection.

Besides data-flow analysis and dependence analysis, one additional program analysis, pointer analysis, is critical to performing all the above optimizations effectively. In most languages, objects of user-defined abstractions are allocated on the heap and shared using pointers, so pointer analysis is required to reason about their behavior. In object-oriented languages, polymorphism and dynamic functions complicate the problem of pointer analysis. Pointer analysis becomes more problematic in languages such as C and C++, which use unconstrained pointers to represent memory as a block of bytes. Although good linear algorithms exist for pointer analysis of stack-allocated variables, the modeling of the heap remains a challenge, with algorithms ranging from linear to double exponential analyses. These problems can be alleviated, if not solved, using abstractions with user-defined annotations.

In summary, conventional compilers try to optimize an input program by transforming both the computation and data organizations of the program. As the entire program is composed of many user-defined abstractions, compilers perform interprocedural program analysis to gather information across function boundaries, and eliminate abstraction boundaries through function inlining when necessary.

For interprocedural analysis (whole program analysis), we need to consider all effects on function parameters, return values, and global variables. When analyzing function calls, different levels of precision can be accomplished with call-strings or assumption sets. For object-oriented programs, the modeling of the heap and the states of objects becomes increasingly important. Data hiding and the encapsulation of object states allows modular program analysis, but in general, the scalability of an analysis for real-world applications must be addressed by storing results of previous analysis passes. This becomes particularly important when libraries are used.

In the next section, we describe annotation languages that are suitable for describing properties of user-defined abstractions. Specifically, the results of an analysis can be represented as annotations, and users themselves can provide additional information about the semantics of an abstraction. We focus on specific properties that are required to ensure the correctness of transformations. These annotations enable cross-abstraction optimizations without eliminating the abstraction boundary.

III. Annotating User-defined Abstractions

In most languages that support abstractions in user-defined types, the abstractions can be separated into two categories: function abstractions and data abstractions. Additionally, languages such as C++ support object-oriented abstractions, where different function and data abstractions can relate to each other through subtype relations and through inheritance.

Function (or procedure) abstractions represent algorithms that operate on data. The operations might be as simple as returning the value of a field within a compound data structure, or as complex as sorting elements in a container. Their semantics can be expressed in terms of what restrictions the input data must satisfy before entering the operation, what data are being modified by the operation, and what properties and relations the resulting data would satisfy after the operation.

Data abstractions are encapsulated collections of values that relate to each other. The semantics of data abstractions are normally expressed in terms of properties and invariants that must be satisfied by the data stored in the abstraction. For example, in a singly-linked list, pointers connecting elements must be acyclic. Because implementation details of abstractions are not visible to the outside, such properties can often be described in terms of abstract attributes of the abstraction. These attributes are abstract in that they do not necessarily have concrete storage in the abstraction.

Figure 4 shows the grammar and examples of some annotations that we developed for optimizing loops that operate on user-defined array abstractions [22]. The annotations in Figure 4 are preliminary and need to be extended in many ways. However, as shown in the following, these annotations serve as informative examples to illustrate what semantics need to be described by a complete annotation language.

A. Function Annotations

In Figure 4(b), all annotations except (1) and (5) are function annotations. These annotations describe semantics of functions that operate on the $floatArray$ and $Range$ data abstractions, which in turn are described in (1) and (5). The semantics of these functions can be separated into the following categories.

- *Restrictions on the inputs of the operation*. In Figure 4(a), three annotations, $read$, $allow\text{-}alias$, and $restrict\text{-}value$ are used to describe input data of a function abstraction. As shown in examples (2), (4), (6) and (8) in Figure 4(b), $read$ lists all the memory locations that are accessed by the operation; $allow\text{-}alias$ describes restrictions on aliasing relations between locations of the input data—everything not listed in $allow\text{-}alias$ cannot be aliased with other inputs; $restrict\text{-}value$ describes relations between values of input data. Note that $restrict\text{-}value$ can also be used to describe relations between input data and results, as illustrated in examples (6) and (8).
- *Modification side-effects*. In Figure 4(a), the $modify$ annotation describes modification side effects, specifically, what data (variables) are modified by the operation.

```
<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

```
(1) 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); };
(2) operator floatArray::operator =
(const floatArray& that):
modify-array (this) {
    dim = that.dim; len(i) = that.len(i);
    elem(i$x:1:dim) = that.elem(i$x); };
(3) operator +(const floatArray& a1,double a2):
new-array () { dim = a1.dim; len(i) = a1.len(i);
          elem(i$x:1:dim) = a1.elem(i$x)+a2; };
(4) operator floatArray::operator ()
(const Range& I):
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); };
(5) class Range: has-value {stride; base; len; };
(6) 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;};};
(7) operator floatArray::operator() (int index) :
inline { this.elem(index) };
restrict-value { this = { dim = 1; };};
(8) 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

The items listed by $modify$ must include all memory storage reachable from the function parameters and global variables.

- *Relations between results and inputs*. In Figure 4(a), the annotations $new\text{-}array$, $modify\text{-}array$, $restrict\text{-}value$, and $alias$ can all be used to describe relations between the results and the inputs of an operation. The annotations $new\text{-}array$ and $modify\text{-}array$ are specific to array abstractions. The $restrict\text{-}value$ annotation describes relations between values of inputs and results. The $alias$ annotation describes the aliasing relations between inputs and results.
- *Rewrite annotations*. In Figure 4, the $inline$ annotation is essentially a transformation directive that eliminates abstraction boundaries. It describes an operation by replacing it with a collection of equivalent operations. The $inline$ annotation therefore can be seen as a transformation specification for rewriting function abstractions.

B. Data Annotations

In Figure 4(b), examples (1) and (5) describe the semantics of user-defined data abstractions. Specifically, they describe properties of the $floatArray$ and $Range$ classes. These properties can be separated into the following categories.

- *Data attributes*. In Figure 4(b), example (1) uses the $is\text{-}array$ annotation to specify that the $floatArray$ class has three data attributes: dim, len and $elem$, where dim is a single scalar value, and len and $elem$ are collections of values that are indexed by a sequence of integer parameters. Similarly, example (5) uses the $has\text{-}value$ annotation to specify that the $Range$ class has three data attributes: $stride$, $base$ and len, where all the attributes are scalar values. Data attributes conceptually model the values stored within a data abstraction. However, internally the values may be implicitly represented in various forms and concrete storage may not be found for the specified attributes.
- *Properties of attributes and relations among them*. In Figure 4(b), when example (1) uses $is\text{-}array$ to describe $floatArray$, it implicitly conveys that $floatArray$ has Fortran90 array semantics; that is, the dim and len attributes describe the shape of the array, the $elem$ attribute returns the elements within the array, and no elements of the array are aliased. Similarly, the attributes described in example (5) for the $Range$ class must satisfy $len \geq 0$. The $allow\text{-}alias$ and $restrict\text{-}value$ annotations in Figure 4 need to be extended to describe such properties. Figure 3 shows some of the extensions to these annotations.
- *Relation between attributes and functions*. In Figure 4, each attribute declaration must be followed by a defini-

tion, which includes function calls necessary to extract the attribute values from the data abstraction. Furthermore, functions that operate on data abstractions are described in terms of their effects on the attributes. Examples of such annotations are discussed in more detail in Section III-A.

- *Rewrite annotations*. In Figure 4, the $is\text{-}array$ annotation specifies a collection of definitions that can be used to replace an array abstraction under certain conditions. Specifically, we can replace the array abstraction with a more efficient equivalent implementation. Such annotations are very similar to the $inline$ directive in example (7) and are used solely for optimization purposes.

C. Object-oriented annotation

In object-oriented languages, user-defined abstractions can inherit from each other and have sub-type relations among one another. For example, in C++, a derived class can adapt the behavior of its superclasses by re-implementing virtual functions. An abstraction annotation language therefore must model relationships between different abstractions. In Figure 4(b), the $inheritable$ annotation in example (1) specifies whether the array semantics described by the $is\text{-}array$ annotation can be inherited by subclasses of $floatArray$. In general, an annotation language needs to support not only the inheritance of various semantic properties, it must also provide mechanisms to specify how behaviors of abstractions are adapted by inheritance.

D. Discussion

The annotations in Figure 4 are far from becoming a complete annotation language, which is the topic of our future research. This section uses our prior experience to summarize and to speculate on what properties of user-defined abstractions need to be described by an annotation language in order to significantly extend the applicability of compiler analysis and optimizations.

An annotation language is not complete unless we can verify that the specified properties reflect the implementations of abstractions. Otherwise, misinformed annotations would lead compilers to generate incorrect programs. We believe that identifying what properties need to be annotated is a significant step toward verifying such properties, as program analysis often needs external annotations from programmers to be effective. Our future research includes both identifying additional annotations that would benefit compiler optimizations and developing program analysis techniques to verify such annotations.

IV. Extending Traditional Optimizations

Conventional compilers often fail to apply the optimizations discussed in Section II to codes that use user-defined abstractions due to the lack of information about the semantics of these abstractions, which are often obscured from traditional compiler analysis for the following reasons.

1) Compiler analysis must conservatively approximate behaviors of the input program (*e.g.*, due to lack of runtime information) so that the abstraction properties required for an optimization cannot be established.
2) In order to establish useful properties of user-defined abstractions, compilers must perform interprocedural analysis. But programs typically use many libraries, making whole-program analysis too expensive to be practical.
3) The source code of a library may not be available to the compiler, and the compiler cannot extract the required information from the library's object code.

All three problems can be addressed with annotations as follows.

1) Since the programmer is aware of the semantics of abstractions that he defines, he can explicitly specify properties of his abstractions using annotations.
2) A separate library analysis can generate the annotations as each library is processed (compiled). The annotations can be saved and serve as external descriptions of the properties of user-defined abstractions.
3) If no source code of a library is available, it suffices to annotate the library's interface and make such annotations available to optimizations of the application (that uses the library). This solution does not permit inlining of library code (when not available from the C++ header files).

Using the annotations discussed in Section III, we can extend traditional program analysis with high-level semantics of user-defined abstractions. After integrating the additional information from annotations, the extended analysis is "aware" of the specific properties of user-defined abstractions.

An *abstraction-aware analysis* (A^3) uses information specified by external annotations to combine the semantics of built-in types with the properties of user-defined abstractions. Specifically, it computes more precise information regarding program behavior to facilitate more advanced traditional optimizations. Consequently, the annotations have bridged the semantic gap due to imprecision of conservative static analysis, impractical scalability of whole-program analysis, and the lack of a sufficient high-level representation of libraries (missing source code).

When program analysis becomes abstraction-aware through external annotations, traditional optimizations are naturally extended accordingly to be abstraction-aware. For example, in our running example in Figure 3, the function $eval$ is annotated to indicate that it does not modify the variables a and $this$ and that it creates no aliasing. Thus, calling this function does not change the object's state. This information is crucial for traditional analysis such as *available expressions analysis* or *very busy expression analysis*. In our approach, we extend the analysis to include such function calls that do not change the object's state in in the collection of available expressions. Based on the results of available expressions analysis, redundant expressions can be eliminated. Hence, we

Optimization	A1	A2	A3	A4
Common subexpression elimination	no	no	no	no
Loop transformations	no	no	no	no
Procedure inlining	no	yes	yes	no
Structure splitting	no	yes	no	yes
AA Scalar replacement	yes	no	yes	yes
AA Loop transformations	yes	no	yes	yes
AA Common subexpression elimination	yes	no	yes	no
OpenMP parallelization of container iteration	yes	no	yes	yes
Iteration-space narrowing	yes	no	yes	no
Iteration-space partitioning and loop specialization	yes	no	yes	no
Iteration-space tiling	yes	no	yes	yes
Precomputation	yes	yes	yes	no
Array abstraction translation	yes	yes	no	yes
Elimination of indirection	yes	yes	yes	yes

TABLE I

OPTIMIZATIONS CLASSIFIED IN FOUR DIFFERENT ASPECTS

can similarly perform library-aware redundancy elimination on applications that use libraries.

Table I enumerates a collection of optimizations that we have identified as crucial for improving the performance of scientific applications used within the DOE laboratories, several of which are described by White, *et al.* [19]. This optimization classification table shows that we have identified different categories of optimizations, defined by the combination of the following aspects of an optimization.

A1. Does the optimization require high-level semantics of the user's abstraction (first column)?
A2. Does the optimization eliminate abstraction layers/boundaries? (second column)
A3. Does the optimization apply to function abstractions? (third column) For example, procedure inlining eliminates a function abstraction. Note that this optimization applies to the application code, not the library (the function remains in the library).
A4. Does the optimization apply to data abstractions or use semantics of data abstractions? (fourth column) For example, when applying structure splitting, the data abstraction of the original structure is eliminated and replaced by new data structures.

A special category in Table I, neither requires high-level abstraction semantics nor eliminates abstraction layers. This category includes most of the optimizations in conventional compilers, such as common subexpression elimination, scalar replacement, and loop transformations.

Three optimizations are explicitly denoted as Abstraction Aware (AA) optimizations. For these optimizations, we have found that their corresponding traditional optimizations, scalar replacement, loop transformations, and common subexpression elimination without utilizing annotations, are essential as well in improving the performance of our applications. Other optimizations such as the OpenMP parallelization is based on the high-level semantics of STL containers and the iterators defined for it [13].

Most conventional compilers implement optimizations that do not require elimination of any abstractions, compilers. Some also implement non-trivial optimizations that does not require high-level semantics of abstractions but eliminates user-defined abstractions, such as structure splitting. In contrast, all the abstraction-aware optimizations require annotations, especially when the required properties cannot be established by a conventional program analysis. Even more advanced transformation capabilities are required for a compiler infrastructure to permit optimizations such as array abstraction translation, which we have demonstrated using ROSE [14].

We also identified optimizations for which we did find useful abstract-aware extensions. Those optimizations are mostly applied in the back-ends of compilers. Examples of such optimizations include *code selection*, which can be performed by Bottom Up Rewrite Systems (BURS) with tools such as burg or iburg, but are not useful at the AST level in our experience because burg requires absolute values as weights for the selectable instructions.

V. ROSE INFRASTRUCTURE

We are implementing our work on optimizing user-defined abstractions within ROSE, a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure for optimizing large-scale (on the order of a million lines or more) DOE applications [14], [16]. The ROSE framework enables tool builders who do not necessarily have a compiler background to build their own source-to-source optimizers. The current ROSE infrastructure can process C and C++ applications, and we are extending it to support Fortran90 as part of on-going work.

The ROSE infrastructure provides several components to build a source-to-source optimizer. A complete C++ front-end is available that generates an object-oriented abstract syntax tree (AST) as an intermediate representation. Optimizations are performed on the AST. The AST preserves the high-level C++ language representation so that no information about the structure of the original application (including comments) is lost. A C++ back-end can be used to unparse the AST and generate C++ code. The user builds the "mid-end" to analyze or transform the AST, and ROSE assists by providing a number of mid-end components, including a predefined traversal mechanism, an attribute evaluation mechanism, transformation operators to restructure the AST, and predefined optimizations. ROSE also provides support for library annotations whether they be contained in pragmas, comments, or separate annotation files.

A. Front-End

We use the Edison Design Group C++ front-end (EDG) [7] to parse C and 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++ [3]. SAGE III is used by the mid-end as an intermediate representation. Full template support permits all templates to be instantiated, as required, in the AST. The AST passed to the mid-end represents the program and all the header files included by the program. The SAGE III IR has 240 types of IR nodes, as required to represent fully the original structure of the application in the AST.

B. Mid-End

The mid-end permits the analysis and restructuring of the AST for performance improving program transformations. Results of program analysis are attached to AST nodes. The AST processing mechanism computes inherited and synthesized attributes on the AST. 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.

ROSE internally implements a number of forms of procedural and interprocedural analysis. Much of this work is in current development. ROSE current includes support for dependence, call graph, and control flow analysis.

To support whole-program analysis, ROSE has additional mechanisms to store analysis results persistently in a database (*e.g.*, SQLite or MySQL), to store ASTs in binary files, and to merge multiple ASTs from the compilation of different source files into a single AST (without losing project file and directory structure).

C. Back-End

The back-end unparses the AST and generates C++ source code. Either all included (header) files or only source files may be unparsed; 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. Full template handling is included with transformed templates output in the generated source code.

VI. RELATED WORK

Several projects address optimization of user-defined abstractions, including our own prior work on loop optimizations for array abstractions [14], [22]. Among these, our goals are most similar in spirit to the Broadway compiler by Guyer and Lin [8]. They develop a specific annotation language for Broadway, and then apply this work to optimize the use of C libraries written in an object-oriented style. In this paper, we consider a broader set of possible annotations that can directly express relationships between function and data abstractions, as well as the unique characteristics of object-oriented languages.

Other compiler projects have also placed significant emphasis on optimizing libraries, especially in the general context of *Telescoping Languages* [10]. The SUIF compiler [15], MAGIK compiler [5], and MPC++ [4], [9] each provide a programmable level of control over the compilation of applications in support of library optimizations, but require users to implement transformations within the compiler itself. By contrast, Schupp, *et al.*, develop for C++ an expression simplifier, which users can extend for optimizing their own abstract data types through annotations, inserted in a C++-style syntax directly into the application [17]. Similarly, CodeBoost (for C++) allows its users to assign simple tags to variables in the source that can be interpreted by its rule-based rewrite system [2]. We share the design goal of developing annotations which may be provided by library developers who do not have a compiler background.

Annotations, computed or specified off-line, are particularly critical during run-time optimization. Krintz and Calder use annotations to reduce the run-time cost due to analysis when dynamically optimizing Java programs [12], and Grant, *et al.*, use annotations similarly to guide specialization in their dynamic optimizer for C [6]. Although we do not pursue dynamic optimization here, these projects emphasize the importance and utility of annotations as an interface between the analysis and optimizer.

We have not specified a particular annotation language in this paper, but expect to draw from existing examples used by the Broadway compiler, and possibly from more formal logic-based systems used in verification [11].

Template Meta-Programming can also optimize user-defined abstractions [18], but is effective only when optimizations are isolated within a single statement. Optimizations such as loop fusion across statements, which requires dependence analysis, is beyond the capabilities of template meta-programming.

Prior work has also applied high-level optimizations to user-defined abstractions. Specifically, Wu, Midkiff, Moreira and Gupta [20] proposed *semantic inlining*, which allows their compiler to treat user-defined abstractions as primitive types in Java. Artigas, Gupta, Midkiff and Moreira [1] 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 [21] investigated automatic parallelization of loops operating on user-defined containers, but assumed that their compiler knew about the semantics of all operators. These approaches encode knowledge within their compilers, and thus 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 communicate explicitly with the compiler.

VII. CONCLUSIONS AND FUTURE WORK

This paper discusses the features of an annotation language that we believe to be essential for optimizing user-defined abstractions. These features should capture semantics of function, data, and object-oriented abstractions, express abstraction equivalence (*e.g.*, a class represents an array abstraction), and permit extension of traditional compiler optimizations to user-defined abstractions. Our future work will include developing a comprehensive annotation language for describing the semantics of general object-oriented abstractions, as well as automatically verifying and inferring the annotated semantics.

References

[1] P. V. Artigas, M. Gupta, S. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.

[2] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Proc. Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, September 2003. IEEE Computer Society Press.

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[4] S. Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[5] D. R. Engler. Incorporating application semantics and control into compilation. In *Proc. USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, USA, October 1997.

[6] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1997.

[7] E. D. Group. http://www.edg.com.

[8] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, Berkeley, CA, Oct. 3–5 1999. USENIX Association.

[9] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++—MPC++ approach. In *Proc. Reflection '96 Conference*, April 1996.

[10] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *J. Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.

[11] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *Proc. OOPSLA*, Seattle, WA, USA, November 2002.

[12] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proc. PLDI*, Snowbird, UT, USA, June 2001.

[13] D. Quinlan, M. Schordan, Q. Yi, and B. de Supi nski. Semantic-driven parallelization of loops operating on user-def ined containers. In *16th Annual Workshop on Languages and Compilers for Parallel C omputing*, Lecture Notes in Computer Science, Oct. 2003.

[14] D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen. Classification and utilization of abstractions for optimization. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, October 2004.

[15] M. S. L. 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.

[16] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[17] S. Schupp, D. Gregor, D. Musser, and S.-M. Liu. User-extensible simplification—type-based optimizer generators. In *Proc. International Conference on Compiler Construction*, volume LNCS 2027, pages 86–101, Genova, Italy, April 2001. Springer-Verlag.

[18] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[19] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proc. International Conference on Supercomputing*, Boston, MA, USA, June 2005.

[20] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Improving Java performance through semantic inlining. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, Mar 1999.

[21] P. Wu and D. Padua. Containers on the parallelization of general-purpose Java programs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Oct 1999.

[22] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under contract W-7405-Eng-48.