LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-202762

Applying Loop Optimizations to Object-oriented Abstractions Through General Classification of Array Semantics

Qing Yi, Dan Quinlan

March 8, 2004

The 17th International Workshop on Languages and Compilers for Parallel Computing
West Lafeyette, IN, United States
September 22, 2004 through September 25, 2004

Applying Loop Optimizations to Object-oriented Abstractions Through General Classification of Array Semantics

Qing Yi Dan Quinlan

Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

Abstract. Optimizing compilers have a long history of applying loop transformations to C and Fortran scientific applications. However, such optimizations are rare in compilers for object-oriented languages such as C++ or Java, where loops operating on user-defined types are left unoptimized due to their unknown semantics. Our goal is to reduce the performance penalty of using high-level object-oriented abstractions. We propose an approach that allows the explicit communication between programmers and compilers. We have extended the traditional Fortran loop optimizations with an open interface. Through this interface, we have developed techniques to automatically recognize and optimize user-defined array abstractions. In addition, we have developed an adapted constant-propagation algorithm to automatically propagate properties of abstractions. We have implemented these techniques in a C++ source-to-source translator and have applied them to optimize several kernels written using an array-class library. Our experimental results show that using our approach, applications using high-level abstractions can achieve comparable, and in cases superior, performance to that achieved by efficient low-level hand-written codes.

1 Introduction

As modern computers become increasingly complex, compilers must extensively optimize user applications to achieve high performance. One important class of such optimizations includes loop transformations, such as loop blocking and fusion/fission, which have long been applied to Fortran scientific applications.

Most loop optimizations, however, have been applied only to loops operating on primitive types in Fortran or C. To illustrate this, consider Figure 1, a C++ fragment that uses a high-level array class library. Here the user-defined types, $floatArray$ and $Range$, have similar semantics to the Fortran90 array and subscript triplet respectively.

If Figure 1 were written in Fortran90 using the primitive array types, most Fortran compilers would be able to translate the array operations into explicit loop computations and then apply loop optimizations. However, as the same computation is written in C++ using abstractions, a C++ compiler will likely consider all the array operations as opaque function calls, and apply no optimizations.

```
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;
}
```

Fig. 1. Example: 1D interpolation

To avoid this performance penalty for using object-oriented abstractions, programmers are often forced to write low level code and unnecessarily expose many implementation details. This practice discourages code reuse and thus leaves good programming styles in conflict with high performance. The problem is especially acute in scientific applications where performance is critical.

We propose an approach which encourages programmers write high-level object-oriented programs by allowing them to explicitly communicate with the compiler. Our work does not apply to just-in-time compilation techniques, but can apply to Java in a source-to-source manner similar to the way which we apply it to C++. Specifically, we present the following new results.

- We have designed an annotation language specifically for classifying the semantics of user-defined types and operators.
- We have developed a new algorithm, adapted from the traditional constant-propagation algorithm, to propagate semantic properties of object-oriented abstractions. The extracted properties are then used to drive the optimization of user-defined abstractions.
- Based on our annotation interface, we have applied loop optimizations directly to general user-defined array operations. In contrast, traditional loop optimizations only addressed loops operating on primitive language constructs, and not on user-defined abstractions.
- We have implemented the above techniques in a C++ source-to-source translator and have applied them to optimize several kernels written using an array-abstraction library. These results were not possible using traditional techniques for Fortran applications. Our results show that using our approach, applications using high-level abstractions can achieve comparable or even better performance than that achieved by lower-level codes.

Although we promote communication between programmers and compilers, we do not exclude automating the process. Our future work includes at least partially automating the generation of annotations. Section 2 describes our extended loop optimization algorithm. Section 3 and 4 then present our annotation language and overall optimization framework respectively. Finally, Section 5, 6 and 7 present our experimental results, discuss related work, and draw conclusions.

2 Extended Loop Transformation

Figure 2 summarizes our extended loop transformation algorithm, which implements three loop optimizations: interchange, fusion and blocking. The algorithm

```
loop-transformation(C, A)
   C: input code;   A: array abstraction interface
Dep = construct-dependence-graph(C,A);
distribute-loop-nests( C, Dep);
for (each loop nest l ∈ C )
   apply-loop-interchange(l, Dep);
apply-loop-fusion( C, Dep);
for (each loop nest l ∈ C)
   apply-blocking( l, Dep)

construct-dependence-graph(C, A)
   C: input code;   A: array abstraction interface
for (each pair of statements s1 and s2)
  (mod1,use1) = get-side-effects(A,s1);
  (mod2,use2) = get-side-effects(A,s2);
  if ( mod1 or mod2 contains unknown side-effect)
     create-dependence(s1, s2); continue;
  for (each (a1, a2) ∈ {(mod1, use2) or (use1, mod2)
                        or (mod1, mod2) })
    if (is-var(a1) and is-var(a2) and identical(a1,a2))
       create-dependence(s1, s2);
    else if (((arr1,sub1) = is-array-elem(A, a1))
         and ((arr2, sub2) = is-array-elem(A,a2))
         and identical(arr1,arr2))
       compute-array-dependence(sub1, sub2);
    else if (may-alias(A,a1, a2))
      create-dependence(s1, s2);

is-array-elem(A, a1)
     A: array abstraction interface;
     a1: memory access expression;
     return: array and subscripts of the access
 if ((obj, subs) = A.is-access-array-elem(a1))
   return (obj, subs);
 else if ((pntr,subs) = is-pntr-array-access(a1))
         and is-constant-pointer(pntr))
    return (pntr, subs);
 else return ∅

may-alias(A, a1, a2)
     A: array abstraction interface;
     a1,a2: memory references to be analyzed;
     return : whether a1 and a2 may be aliased;
 if ((arr1, sub1) = A.is-access-array-elem(a1))
    return may-alias(arr1, a2);
 if ((arr2, sub2) = A.is-access-array-elem(a2))
    return may-alias(a1, arr2);
 if (A.is-known-array(a1) and
     A.is-known-array(a2))
     return A.is-aliased-array(a1, a2);
 if (a1 and a2 are both local variables)
     return is-aliased-local-var(a1, a2);
 else
     return is-type-alias-compatible(a1,a2);
```

Fig. 2. Loop transformation algorithm

is an extension to previous work by Yi, Kennedy and Adve [22], which optimized loops in Fortran applications. We have extended their dependence analysis algorithm with an interface, *array abstraction interface*, to facilitate the optimization of loops operating on user-defined array classes [1].

2.1 Dependence Analysis

In Figure 2, function *construct-dependence-graph* computes the reordering constraints between each pair of statements (s_1,s_2). Specifically, a transformation is safe if it never reorders any iterations of s_1 and s_2 that are connected by dependences.

In the algorithm, we first collect all the memory references modified (mod_1 and mod_2) or used (use_1 and use_2) by s_1 and s_2 respectively. We then create dependence edges to connect each pair of iterations, $I_1(s_1)$ (iteration I_1 of statement s_1) and $I_2(s_2)$, when both $I_1(s_1)$ and $I_2(s_2)$ may access a common memory store *loc*, and at least one of them may modify *loc*.

Given two arbitrary memory references, a_1 and a_2, if they access memory through an identical scalar or array variable[2], we compute their reordering constraints using traditional dependence analysis algorithms for Fortran [1, 19, 3], as

[1] The array abstraction interface is also used in the profitability analysis of applying loop optimizations, which is omitted in this paper.

[2] Here the variable could be a class variable concatenated with a list of non-pointer field names.

encoded by functions *create-dependence* and *compute-array-dependence* respectively. Note that function *is-array-elem(A,a_1)* uses the array-abstraction interface A to determine whether a_1 accesses the element of some array arr_1 using subscripts sub_1. Section 2.2 describes this function in more detail.

If a_1 or a_2 indirectly access memory through dereferencing unknown address pointers, or if they refer to different variables, we perform aliasing analysis to determine whether a_1 and a_2 may reach a common memory store. Unless a_1 and a_2 can be proved to never alias to each other, we connect them with a dependence edge to disable optimizations that attempt to reorder them.

2.2 Array Abstraction Interface

In object-oriented languages such as C++, programmers can define their own array classes. As the addresses of class objects cannot be redefined, these array classes can be treated as if they are Fortran arrays, and their aliasing relations can be determined more easily than C pointers.

We use an array-abstraction interface to communicate the semantics of user-defined array types with our loop optimizer. In Figure 2, we use this array-abstraction interface to both recognize user-defined array objects and to determine the aliasing relations between array objects.

We use function *is-array-elem(A,a_1)* to query the array-abstraction interface A whether a_1 is a subscripted array element access. If yes, we return both the array object *obj* and a list of integer subscripts *subs* (multi-dimensional array is allowed). Otherwise, we determine whether a_1 is the subscript operator for C pointers and whether the address of the pointer is never changed within the optimization scope. If neither cases apply, we conclude that a_1 does not access array elements and return $\emptyset$.

We use function *may-alias(A, a_1,a_2)* to determine the aliasing relations between two memory references a_1 and a_2. First, if either a_1 or a_2 is a subscripted array element access, we examine the corresponding arrays for aliasing relations. Second, if both a_1 and a_2 are array class objects, we query the array-abstraction interface for their aliasing relations. Third, if both a_1 and a_2 are local variables, we use a simple context-insensitive algorithm (*is-aliased-local-var* in Figure 2) to determine whether the input code C has performed operations that might cause a_1, a_2 to reach to a common memory store. Finally, if none of the above cases apply, as long as the types of a_1 and a_2 are compatible, we conservatively assume that they might be aliased.

3 Annotating Semantics of Abstractions

Figure 3(a) shows the grammar of our annotation language. Figure 3(b) shows some example annotations for the 1D interpolation code in Figure 1. Section 3.1, 3.2, and 3.3 will describe the semantics of these annotations in more detail. Section 3.4 then will discuss the inheritance of these annotations. Although we currently require programmers to explicitly annotate all the abstractions, our

```
<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); };
has-value {dim; len$x:0,dim-1=this.getLength(x); }
(2) operator floatArray::operator =
(const floatArray& that):
modify-array (this) {
    dim = that.dim; len(i) = that.len(i);
    elem(i$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. 3. Annotation language

future work will target developing compiler techniques to automate the process. Section 3.5 discusses the automation issues in more detail.

3.1 Array Annotation

We provide three annotations, *is-array*, *modify-array*, and *new-array*, to describe the array abstraction semantics of both user-defined types and operations.

The declaration(1) in Figure 3(b) uses *is-array* to declare that the class $floatArray$ satisfies the pre-defined array semantics. Specifically, it has 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 $i\$x : 0 : dim-1$ denotes a list of parameters, i_1,i_2,...,i_{dim-1}.

In Figure 3(a), the *is-array* annotation also allows a optional "*define <stmts>*" phrase to define loop-invariant operations that should be executed before enumerating array elements. Further, programmers can use *inheritable* to specify that the annotation is preserved by class inheritance.

The declaration(2) in Figure 3(b) uses *modify-array* to declare that the operator "*floatArray::operator= (const floatArray& that)*" performs element-wise modification of the current array (the "this" argument of the C++ member function call). Specifically, the operator first modifies the current array to have

the same shape as the input array *that*. It then modifies each element of the current array to be a copy of the corresponding element in *that*.

The declaration(3) in Figure 3(b) uses *new-array* to declare that the operator "*+(const floatArray& a_1, double a_2)*" 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 declaration(4) declares that the operator "*floatArray::operator()(const Range& I)*" constructs a new array that is aliased to the current one, by selecting only those elements that are within the iteration range I.

3.2 Property Annotation

We use two annotations, *has-value* and *restrict-value*, to describe properties of user-defined abstractions. These properties are described using symbolic values.

The annotation *has-value* declares that a user-defined type has certain properties that can be represented as symbolic values. For example, the declaration(1) in Figure 3(b) uses *has-value* to declare that the class *floatArray* has two properties: the array dimension and the length of each dimension i (if the length cannot be statically determined, $this.getLenth(i)$ will be used in place). Similarly, the declaration(5) declares that the *Range* class (which selects a subset of elements in an array) has three properties, *base*, *len* and *stride*.

The annotation *restrict-value* describes how properties of user-defined types can be implied from function calls. For example, the declaration(6) in Figure 3(b) declares that if "*floatArray::operator()(int index)*" is used to access the element of a *floatArray* object *arr*, the array object *arr* must have a single dimension, and it will remain single-dimensional until some other operator modifies its shape.

3.3 Side-effect Annotation

We provide four annotations, *mod*, *read*, *alias*, and *inline*, to describe the general side-effects of user-defined operators.

The *mod* annotation declares a list of memory references that might be modified by a function. Similarly, the *read* annotation declares the list of the references being used, and the *alias* annotation declares the groups of references that might be aliased to each other. These annotations communicate with our alias and side-effect analysis algorithms in resolving semantics of function calls. The declaration(8) in Figure 3(b) shows an example of using these annotations.

The *inline* annotation declares the high-level semantic interpretations of user-defined functions. As example, the declaration(7) in Figure 3(b) declares that "*floatArray::operator()(int)*" is semantically equivalent to a subscripted element access of the current $floatArray$ object.

3.4 Inheritance of Semantic Annotations

We have provided two class annotations, *is-array* and *has-value*, to describe the properties of user-defined types. Since the *has-value* annotation does not make

```
apply-optimization( C, A)
      C: input code fragment; A: annotation interface
(1) translate element-wise array operations
    (1.1) rewrite-inline-operators( C, A)
    (1.2) valmap = ∅;   property-propagate(C, A, valmap);
    (1.3) rewrite-modify-array(C, A, valmap);   rewrite-new-array(C, A, valmap);
(2) loop-transformation( C, A);
(3) rewrite-array-access( C, A);
```

Fig. 4. Steps of optimizing array abstractions

implicit assumptions about the declared properties, the annotation is preserved by class inheritance. However, the *is-array* annotation assumes that elements of an array object cannot be aliased, an assumption which can be violated by the derived classes. Consequently, we decide that the derived classes do not automatically inherit the *is-array* annotations unless programmers explicitly specify otherwise (using the *inheritable* annotation). For pointer variables of non-inheritable array types, we first try to precisely determine their types, and if not successful, conservatively assume that their semantics are unknown.

Similar situations arise for virtual functions, whose side-effect annotations (*modify-array*, *new-array*, *restrict-value*, *mod*, *read*, *alias*, and *inline*) should not be inherited. Consequently, we assume that the semantics of virtual function calls are unknown unless their implementations can be statically determined.

3.5 Automatic Extraction and Verification of Annotations

Our annotation language serves as an interface between program analyses and transformations. Though currently annotations can only be manually produced by programmers, our future work will try to automate the process. Automatically extracting semantic annotations is in general a hard problem, which cannot yet be solved satisfactorily through existing techniques. Specifically, the implementation details of user-abstractions (e.g., book-keeping, performance optimizations, parallelization considerations) can easily obscure the real semantics and force compilers to conservatively assume non-existing dependences.

To illustrate the complexity, consider a smart-object class that performs reference-counting and copy-on-write. The objects of this class are conceptually independent of each other, but any global aliasing analysis algorithm would see the internal sharing of data and would conclude that all such objects can potentially be aliased. To figure out that each modification applies uniquely to a single object, the algorithm must know that data sharing happens only when the reference count of a smart-object has value > 1, a piece of context information mostly ignored by existing algorithms. If the programmer chooses to annotate the smart-object assignment as having no aliasing side-effect, it would be rejected by a verifier. However, if an optimizer chooses to trust the programmer, it will always produce correct code because the annotation conveys precisely the external semantics of smart-objects. Other implementation details can produce similar effects. We are investigating techniques to overcome these issues.

```
property-propagate(C, A, vmap)
  C: input code fragment; A: annotation interface;
  vmap: map from objects to their property values
wklist = ∅;  defUse = build-dataflow-graph(C);
find-restr-op(C,A,vmap,wklist);
while (wklist ≠ ∅)
  cur = pop(wklist);  curval = vmap(cur);
  if (cur is a def-node in defUse)
    for (each cur's unique use-node n)
    if (vmap(n).merge-with(curval) changes)
      find-restr-op(stmt(n),A,vmap,wklist);
  else if (cur has a unique def-node n and
           vmap(n).merge(curval) changes)
    find-restr-op(stmt(n),A,vmap,wklist);
    add n to wklist
```

```
find-restr-op(C, A, vmap, wklist)
  C: input code fragment;
  A: annotation interface;
  vmap: map from objects to their property values
  wklist: working list of object references;
for (each expression exp ∈ C)
  if (desc =A.is-value-restr-op(exp, vmap))
    for (each (ref, value) ∈ desc))
      if (vmap(ref).merge(value) changes)
        add ref to wklist;
```

Fig. 5. Property propagation algorithm

4 Optimizing array abstractions

Figure 4 summarizes the overall steps of our optimizer. Given both the input code and the annotations, we first translate all the collective array operations into an intermediate form of explicit loop computations. Then, we apply the loop optimization algorithm in Figure 2 to the intermediate form. Finally, we translate the output code into efficient low-level implementations.

As shown in Figure 4, we further separate the first step into three sub-steps. First, we replace all the function calls with *inline* annotations with their semantic definitions. Then, we apply an adapted constant propagation algorithm to derive the properties of all objects annotated with *has-value* declarations. Specifically, we try to precisely determine the shapes of all array objects. Finally, we translate all the operations annotated with *modify-array* and *new-array* declarations into explicit loop computations.

It is important that we apply loop optimizations to an intermediate form instead of to the final low-level implementations. An example of the intermediate form is shown in Figure 7, where all the array accesses are explicitly represented as member function calls, and all object properties are readily available. In contrast, after the final step (3) in Figure 4, multi-dimensional arrays are translated into single-dimensional, and C pointers are used in place of the array objects. These implementation details would obscure the original high-level semantics and disable profitable loop optimizations.

4.1 Property Propagation Algorithm

Figure 5 presents our property propagation algorithm. Given an input fragment and the annotations, this algorithm first builds the data-flow graph (def-use chain) and then propagates the properties of user-defined objects on the data-flow graph. The collected properties are stored in *vmap*, which maps object references to their corresponding property values.

```
rewrite-modify-array(C, A, vmap)
   C: input code fragment; A:array annotation;
   vmap: map from objects to their property values
 for (each statement s ∈ C s.t. A.is-mod-arr(s))
  arr = A.get-mod-arr-obj(s)
  (dim,lens) = get-array-shape(vmap,arr)
  ivars = create-new-intvars(dim);
  if (reshape-arr(A,vmap,arr))
    ns = create-member-fcall("reshape",arr,lens);
    insert ns before s;
  rhs = A.get-mod-arr-elem(s,ivars);
  lhs = create-member-fcall("elem", arr, ivars);
  ns = create-assignment( lhs, rhs);
  for (i = 0; i < ivars.size(); + + 1))
    ns = create-loop( ivars[i], 0, lens[i], ns );
  for (each memory reference a ∈ rhs)
    if ( loop-dependent(lhs, a))
      (tmp, ns1) = copy-array-to-tmp(a);
      insert ns1 before s; replace a with tmp
  replace s with ns;
```

```
rewrite-new-array(C, A, vmap)
   C: input code fragment; A:array annotation;
   vmap: map from objects to their property values
 for (each expression exp ∈ C )
   if ((arr,subs) = A.is-access-arr-elem(exp))
     if (A.is-new-arr(arr))
       replace exp with A.get-new-arr-elem(arr,subs);
     else if (! is-variable(arr) )
       (tmp, ns) = copy-array-to-tmp(arr);
       insert ns before exp; replace arr with tmp
   else if ((arr,i) = A.is-access-arr-len(exp))
     if (A.is-new-arr(arr))
       (dim,lens)=get-array-shape(vmap,arr)
       repl = evaluate(lens, i);
       replace exp with repl;
     else if (! is-variable( arr))
       (tmp, ns) = copy-array-to-tmp(array);
       insert ns before exp; replace arr with tmp;
```

Fig. 6. Array operation translation algorithm

Our algorithm relies on a lattice that is nearly identical to that of the traditional constant-propagation algorithm. The depth of the property lattice is three: *top* → *value* → *bottom*, so each property can change its value twice.

Our algorithm is different from constant propagation in three aspects. First, we propagate symbolic values instead of constants. Second, we imply properties from both the modification and usage of an object, so the propagation is bi-directional. Third, because *restrict-value* annotations are independent of control-flow, the result is always correct even before completion.

In Figure 5, we use a working list (*wklist*) to keep track of the object references whose properties need to be propagated. First, we invoke *find-restr-op* to collect the initial known properties for all objects. Each object is added to the working list if its property collection changes. Since each object has a limited number of different properties, and each property can change its value twice, eventually the working list becomes empty and the iteration terminates.

Each object reference *cur* in *wklist* is processed as follows. If *cur* is a definition node in the data-flow graph, we examine each use-node n connected with *cur*. If n does not have any other definition point, we update n to have the same properties as *cur*. Alternatively, if *cur* is a use-node of the data-flow graph and if *cur* has a single definition point n, we update n to have the same properties as *cur*, and then add n to *wklist* to propagate the properties to other use-nodes connected to n. Whenever the property collection of n changes, we invoke *find-restr-op* to accumulate new properties that depend on the updated properties of n. The process iterates until no more updates are necessary.

4.2 Translating Array Operations

In Figure 6, we use functions *rewrite-modify-array* and *rewrite-new-array* to translate collective array operations into explicit loop computations.

```
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);
  for (int _i = 0; _i < (fineGridSize - 3) / 2; _i += 1)
    (fineGrid.elem)(_i * 2 + 2) = coarseGrid.elem(_i + 1));
  for (int _j = 0; _j < (fineGridSize - 3) / 2; _j += 1)
    (fineGrid.elem)(_j * 2 + 1) = (coarseGrid.elem(_j) + coarseGrid.elem(_j + 1)) / 2.0;
}
```

Fig. 7. Example of translating array operations

rewrite-array-access(C, A)
  C: input code fragment; A:array annotation;
  for (each array declaration *decl* in C)
    insert A.get-pre-def(*decl*) after *decl*;
  for (each statement s in C s.t. A.is-reshape-arr(s))
    insert A.get-pre-def(s) after s; replace s with A.get-reshape-def(s);
  for (each expression *exp* in C s.t. A.is-access-arr-elem(*exp*) or A.is-access-arr-len(*exp*))
    replace *exp* with A.get-arr-elem-def(*exp*) or A.get-arr-len-def(*exp*);

Fig. 8. Code generation for array abstractions

The function *rewrite-modify-array* rewrites each operation s that has *modify-array* semantics. First, by querying the annotation interface A, we determine both the array object *arr* being modified and the new value of each array element. We then determine the new shape of *arr* (by querying *vmap*) and insert a new statement to reshape *arr* if necessary. Next, we create a loop nest that assigns each element of *arr* with the correct new value, and if these new values are dependent on the old values of *arr*, we insert a new statement before s to save the old values into a new temporary array. Note that although *copy-array-to-tmp(a)* is inside a loop, array *arr* is copied at most once, and loop dependence analysis is used to determine the necessity of copying. Finally, we replace statement s with the new loop computation.

After applying *rewrite-modify-array*, we now further apply *rewrite-new-array* to rewrite each expression that creates a new array *arr*, accesses an individual element or reads the shape of *arr*, and then discards *arr*. If *arr* is created by an operator annotated with the *new-array* declaration, we avoid creating *arr* by evaluating the information access directly based on the element and shape definitions of the *new-array* annotation. Otherwise, if the semantics of the operator that creates *arr* is unknown, we create a new variable *tmp* to remember the created temporary array and then replace all the other creations of *arr* with *tmp*. By saving the temporary array into a new variable, we create *arr* only once, instead of creating the same array multiple times in the original code.

Figure 7 shows the result of rewriting the 1D interpolation code in Figure 1. Here the array assignment operators, having *modify-array* semantics, are translated into explicit loops. Further, the array *plus* and *division* operators, having *new-array* semantics, are translated into operations on individual elements.

```
void Five_Point_Stencil ( Index & i , Index & j ) {
  Solution (i,j) = ( Mesh_Size * Mesh_Size * Right_Hand_Side (i,j) + Solution (i+1,j)
                   + Solution (i-1,j) + Solution (i,j+1) + Solution (i,j-1) ) / 4.0;
}
void Red_Black_Relax (int gridSize) {
  Index Black_Odd(1,(gridSize - 1) / 2,2),  Black_Even(2,(gridSize - 2) / 2,2);
  Index Red_Odd(1,(gridSize - 1) / 2,2),  Red_Even(2,(gridSize - 2) / 2,2);
  Index Odd_Rows(1,(gridSize - 1) / 2,2),  Even_Rows(2,(gridSize - 2) / 2,2);
  Five_Point_Stencil ( Black_Odd , Odd_Rows );   Five_Point_Stencil ( Black_Even , Even_Rows );
  Five_Point_Stencil ( Red_Even , Odd_Rows );   Five_Point_Stencil ( Red_Odd , Even_Rows );
}
```

Fig. 9. Two-dimensional red-black relaxation

4.3 Generating Low-level Implementations

Figure 8 describes the final step of our algorithm, where we replace array abstraction operations with low-level implementation details. First, as each array abstraction is associated with a list of loop-invariant statements, we insert these statements immediately after the declaration of each array variable and immediately after each operation that reshapes the array variable. Second, we replace all operations that read or modify array objects with low-level implementations.

5 Experimental Results

This section presents our results from optimizing kernels written using the A++/P++ Library [14, 12], 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.

Figures 1 and 9 present the original versions of the single-dimensional interpolation and the two-dimensional red-black relaxation[3] respectively, both using array abstractions. The optimized versions are similar to the 1D interpolation code in Figure 7, except that all the loops are fused and that the final code uses low-level C implementations.

We measured the performance of three versions for each kernel: the original version (*orig*) using array abstractions, the *translate-only* version auto-optimized

[3] In our experiment, the *Five_Point_Stencil* function is inlined within *Red_Black_Relax*.

	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

Table 1. Interpolation results (different numbers of iterations were run for different problem sizes)

	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 2. Red-Black Relaxation Results (different numbers of iterations were run for different problem sizes)

by translating array operations into low level C implementations, and the *translate+fusion* version auto-optimized both with array translation and loop fusion. 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.

Tables 1 and 2 present our measurements using multiple array sizes. Here column *orig* lists the elapsed time spent executing the original versions written using array abstractions; column *translate-only* lists the speedups from translating array abstractions into low-level C implementations (obtained by dividing execution time of the *orig* with that of *translate-only*); column *translate+fusion* lists the speedups from both array translation and loop fusion; column *fusion-only* lists the results from dividing *translate+fusion* columns with *translate-only* columns, i.e., the speedups from applying loop fusion alone.

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

From Table 2, the dominate performance improvements come from translating array abstractions into low-level implementations(*translate-only*). Loop fusion can further improve performance by the factor of 2.3-6.5 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. We are working on better algorithms to selectively apply loop fusion and avoid overly aggressively loop fusion.

The final codes generated by our optimizer are 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 and more reliably applied automatically by compilers than manually by programmers.

6 Related Work

Prior research has developed a rich set of loop transformation techniques [18, 11, 13, 8, 6] for optimizing scientific applications. However, most of these techniques target only explicit loop computations operating on primitive array types, such as the arrays in Fortran or *restrict* pointers in C. In contrast, we target extending these techniques to optimize high-level user-defined types, whose semantics are obscured from the compiler. Our approach could be sufficient for optimizing general object-oriented array abstractions such as the C++ *valarray* and containers in STL, though we do not yet have results for these abstractions.

Previous work has also attempted to apply 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 [2] 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. Quinlan and Schordan [15] developed a C++ compiler infrastructure and used it to translate abstractions from the A++/P++ array library into lower level loops. 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. 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 array abstractions by allowing programmers to explicitly communicate with the compiler.

Several other compiler projects have placed significant emphasis on optimizing libraries, especially in the general context of *Telescoping Languages* [5]. The

SUIF compiler [16], MPC++ (OpenC++) and *MPC++* [10, 7] each provided a programmable level of control over the compilation of applications in support of library optimizations. Other approaches, such as the Broadway compiler [9], uses more general *annotation languages* to guide source code optimizations. However, these frameworks have not focused on applying loop optimizations to object-oriented abstractions.

Template Meta-Programming[17, 4] can also optimize user-defined abstractions, but is effective only when optimizations are isolated within a single statement. Loop fusion across statements, which requires dependence analysis, is beyond the capabilities of template meta-programming.

7 Conclusions

Through the classification of array semantics, this paper develops techniques for applying aggressive loop optimizations to general user-defined abstractions. This work shows promise in significantly raising the level of abstraction while eliminating the associated performance penalty. Although we have only demonstrated our approach on C++ array abstractions, which are similar to F90 array constructs, the approach extends to more general object-oriented abstractions.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.
2. 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.
3. U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.
4. F. Bassetti, K. Davis, and D. Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In I. et al., editor, *International Scientific Computing in Object-Oriented Parallel Environments, ISCOPE 97*, volume 1343 of *LNCS*. Springer, 1997.
5. B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, K. Kennedy, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 2000.
6. S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.
7. S. Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.
8. S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
9. S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. *ACM SIGPLAN Notices*, 35(1):39–52, Jan. 2000.

10. e. a. Ishikawa, Y. Design and implementation of metalevel architecture in c++ - mpc++ approach -. April 1996. San Francisco, USA.
11. M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.
12. M. Lemke and D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In *LNCS*. Springer Verlag, 1992. Proceedings of CONPAR/VAPP V.
13. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
14. D. Quinlan and R. Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
15. D. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In H. G. Dietz, editor, *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Revised Papers*, volume 2624 of *Lecture Notes in Computer Science*, pages 570–578. Springer Verlag, 2003.
16. 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.
17. T. Veldhuizen. Expression templates. In S. Lippmann, editor, *C++ Gems*. Prentice-Hall, 1996.
18. M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, June 1991.
19. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, 1989.
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, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27:219–264, 2004.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.