Anticipatory Optimization in Domain Specific Translation

Ted J. Biggerstaff

September, 1997

Technical Report
MSR-TR-97-22

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Anticipatory Optimization in Domain Specific Translation

Ted J. Biggerstaff
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
tedb@microsoft.com

Abstract

Anticipatory Optimization *(AO) is a method for compiling compositions of abstract components (which are themselves composite data structures such as arrays, matrices, trees, record composites, etc.) so as to anticipate non-optimal structures in the compiled code (e.g., redundant iterations) and to compile the abstract components directly to optimized code without ever producing the non-optimal structures. Anticipatory Optimization is performed in the context of the domain specific operators and operands (i.e., composites) and is driven by the form and semantics of those operators and operands. In contrast, conventional optimizing strategies wait until the domain specific operators and operands (i.e., composites) have been compiled into programming language level code (e.g., C) before they even start the optimization process. By this time, the data flow, variable aliasing, and variable dependency knowledge implicit in the domain specific operators and operands has been lost. So, the first step in conventional optimization is to perform a difficult analysis process to recover this lost information, which will then be used to drive the optimization process. AO avoids such difficulties by using the implicit knowledge to directly generate optimized code.*

Key Words and Phrases: Abstraction, development environments, domain specific, generators, optimization, and reuse.

1. The problem

The fundamental contribution of high level languages (hll-s) to programming productivity was the ability of their compilers to transform a compact representation of a mathematical computation (i.e., an arithmetic expression expressed in a mature notation borrowed from mathematics) into an equivalent but expansive and complexly interrelated series of assembly language instructions. This saves the programmer a significant amount of work because the mathematical expression is simpler, is more natural to write, has fewer constraints, and is more compact than its assembly language equivalent. Other aspects of hll-s have had significantly less impact. For example, hll control constructs are not much more compact nor easier to use than their assembly language counterparts.

Over the years, small incremental improvements to programming productivity have arisen through the addition of new programming constructs (e.g., object oriented constructs, multi-methods, functors, closures, etc.) but none of these have provided as much of a boost as the original innovation of hll expressions. Why? Should not the simple addition of new higher level operators and operands produce exactly the kind of programming productivity improvement seen in the original hll boost? (See Biggerstaff, 1993.) Unfortunately, other factors have complicated the picture. (See Biggerstaff, 1994.) Indeed, adding new higher level operators and operands has improved programming productivity but at the cost of a decline in the performance of the resulting code. Almost without fail, as the level of abstraction has been raised, the performance of the resulting code has declined to unacceptable levels. And this has introduced a serious roadblock to the application of abstraction based strategies for increasing programming productivity. Why?

Figure 1 illustrates the problem. Candidate higher level abstractions (e.g., graphic images) are inherently composite data structures, which require iteration or recursion for their implementation. Further, the operators for such abstractions produce compositions of abstractions that are already inherently composite. Thus, every operation on such abstractions produces a non-atomic, extended computation (i.e., iteration or recursion) because the operands are extended, composite structures. And to compound the problem, the ideal use of such abstract, domain specific operators and operands is in expressions that compose many such operations together. Straightforward translation or interpretation of such compositions of composites leads to highly redundant iteration or recursion.

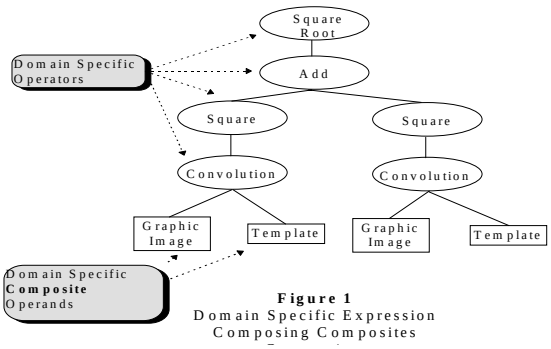

Figure 1
Domain Specific Expression Composing Composites

Consider the example from Figure 1. Straightforward compiling of such an expression could generate six separate passes over the images. In contrast, a human programmer would design only one pass over the images and perform all of the operations within that pass.

The problem is further complicated because conventional modes of compiling compositions preserve the individual operations' atomic boundaries in the compiled code and thereby isolate the portions of redundant code within these bounded chunks of compiled code. Such conventional modes of compiling compositions are based on a "substitute and expand" paradigm in which each individual operation is compiled in isolation (i.e., in a "context free" manner) and is therefore not sufficiently influenced by other operations in the composite during its compilation. Hence, conventional compiling strategies do not produce opportunities for the redundancy in the compiled code to be noticed by the compiler at the compile time nor provide methods for reweaving the compiled code in ways that break the contextual isolation of the neighborhoods induced by the individual high level constructs. Therefore, these conventional compiling methods mostly prevent sharing of redundant code (e.g., sharing the iteration prefixes).

Just to complicate this problem, the boundaries of the redundant portions of the compiled code do not correspond to convenient programming language construct boundaries within the compiled code. That is, to eliminate redundancy, the compiler or post-compilation optimizer might need to merge two loop iteration prefixes (i.e., the "for(*init*; *check*; *update*)" parts of the loop) and manipulate the semantics of the two loop bodies in order to properly connect them. However, such connections depend on the structure and semantics of the iterations, the structure and semantics of the loop bodies, and the context in which the loops occur. Conventional compiling methods are not designed to deal with this kind of code integration.

The conventional alternative to solving this optimization problem is to mop up the redundancy (i.e., reweave the separate chunks of compiled code) after the high level component constructs have been compiled into conventional programming language constructs (and after those high level abstractions have been eliminated by the compiling process). So, conventionally, the implementation level code is rewoven to eliminate the redundancies by conventional optimization strategies[1] such as "loop fusing" and "loop coalescing." However, this post-compilation approach leads to a further complication. A large amount of complex inference (e.g., data flow, dependency, and alias analysis) is usually required to provide the information necessary to accomplish these conventional optimization strategies. Ironically, this approach is "recovering" some of the information that is more directly inferable from the high level abstractions in their pre-compiled forms. Worse, the recovery task introduces the risk of missing legitimate optimizations because of the inadequacy of the analysis tools (e.g., some variable aliasing may be missed). Hence, post-compilation optimization strategies often achieve mediocre results at a great cost in processing time.

Thus, we have dilemma. If we introduce highly abstract operators and operands into our programming notation to increase our programming leverage, we either get poor performance in the resulting code or a chance of marginally acceptable programming performance at the cost of developing extensive, expensive, and inherently incomplete program analysis tools.

2. The Solution: Anticipatory Optimization

The solution is to transform the abstract composite expression into a form that will compile directly into optimal code by using the semantics of the abstract operators and operands, thereby eliminating the need to perform expensive and possibly only partially successful analyses of the compiled code.

The AO method operates as follows: It performs multiple[2] distinct recursive walks of the abstract syntax tree (AST). As it goes, individual transformations are triggered by the pattern of domain specific operators and operands at each subtree. These transformations may

[1] See Bacon, *et al*. 1994. Per conventional usage, "loop fusion" is the merging of two loops at the same level of nesting, and "loop coalescing" is the merging of two loops, one nested within the other. Both depend on fully determining the data flow dependency and aliasing relationships of data items within the body of the loop. The complexity of this analysis arises from, among other reasons, the possibility of data aliasing, i.e., one variable masquerading in non-obvious ways as another variable because of the complexities allowed by the programming language, or a data reference expression referring to a data location whose exact identity requires an inference process.

[2] I will use four passes for this domain example, but other domains could require a fewer or greater number of passes.

rewrite the tree (e.g., transform one abstraction such as an image into a simpler abstraction such as a pixel); add property tag adornments to the AST expressions -- adornments that anticipate how those expressions might be implemented (e.g., an image might be implemented as an array of pixels); migrate the adornments about the tree; and/or merge the adornments in ways that anticipate optimizations in the eventual implementation (e.g., sharing of loop prefixes).

Overall, the various transformations alter the AST in a step by step manner that maps abstractions into programming constructs, anticipates implementation constructs for composites (e.g., loops), and migrates the anticipatory adornments up the AST merging them together (i.e., making anticipatory optimizations) based on the semantic nature of the operators and operands over which they migrate. Once this migration and merging is complete, the optimized code is generated directly from the adorned domain specific expression. I will look at an example of AO operating on a domain specific expression but first, I need to explain a bit about the domain.

3. An Example

3.1 Domain specific definitions

The following concrete example is drawn from the domain of graphics imaging software and is based on an imaging notation specific to that domain called the Image Algebra (IA). (See Ritter, et al. 1990, 1993, 1996.) This domain is just a convenient domain in which to cast examples of the technique because the domain notation is already defined and mature. However, the AO method is not limited to this specific domain. It is applicable to abstract operators and operands from any application domain (e.g., business software, system software, office software, networking, telephony, etc.). Similarly, while the small example will focus on array- and matrix-based implementations, the general AO method can accommodate other implementation composites (e.g., lists, collections, trees, record composites, etc.) through the use of different adornments and transformations.

In the following example, we will use the so-called *backwards graphics convolution operator* $\oplus$ that defines how the matrix-like operand on the right, which in the Image Algebra is called a "Generalized Template" or "Template[3]" for short, is applied to each pixel neighborhood in the image operand on the left hand side. For an image ***a*** and template ***t***, the behavior of this operator is defined as follows:

$$(a \oplus t) \stackrel{\Delta}{=} \{(y, b(y)) : b(y) = \sum_{x \in X} a(x) * t_y(x), y \in Y\}$$

where X and Y are coordinate sets, X being the h-dimensional coordinates for all of the pixels in the image ***a*** and Y the k-dimensional coordinates for all of the pixels in the image resulting from the operation (shown as the image ***b*** in the definition). t_y is a function which maps $Y \rightarrow (X, F)$ where F is a mathematical field giving the weights to be multiplied with the pixel values in the matrix neighborhood of the *current pixel* of ***a***. In the examples of IA template matrices that I will show, only the F values are explicitly shown. The mapping of the Y coordinates into the X coordinates is implicit in the physical geometry of the matrix but is not explicitly defined. In general, this coordinate mapping is more complex than implied by these simple examples but for the purposes of sketching the implementation, simple examples will be adequate.

So let us look at an example template -- one that performs (part of) Sobel edge detection in an image.

$$t_y = \begin{bmatrix} -1 & \phi & 1 \\ -2 & \Diamond & 2 \\ -1 & \phi & 1 \end{bmatrix}$$

where the diamond indicates the reference point in t_y that corresponds to the current pixel in the image with which t_y is being convolved. This means that for any pixel ***[y1, y2]*** in the image ***a***, the summation in the definition should produce the following expression as the value for pixel ***b[y1, y2]*** in the result:

(a[y1-1, y2 -1] * (-1) + a[y1, y2 -1] * (-2) + a[y1+1, y2 -1] * (-1) +

a[y1-1, y2 + 1] *1 + a[y1, y2 + 1] * 2 + a[y1+1, y2 +1] * 1).

So, given these semantics for the $\oplus$ operator, let us look at how we use the knowledge of these semantics to directly generate optimized code. The example follows the step by step reduction of the Image Algebra statement for performing Sobel edge detection:

$$b = \left[(a \oplus s)^2 + (a \oplus s')^2 \right]^{1/2}$$

where

- ***a*** is a gray-scale image (of dimensions mXn) in which we hope to detect edges,
- ***s*** and ***s'*** are generalized IA *templates* that will be applied to every pixel of ***a***,

[3] Not to be confused with the term "template" as used in programming languages like C or C++.

- $\oplus$ is the *generalized backward convolution operator*[4] that performs the application of the templates ***s*** and ***s'*** to all pixels in ***a***,
- + is the conventional plus[5] operator (on the mathematical field induced by the pixel's fundamental type) applied to corresponding pixels in $(a \oplus s)$ and $(a \oplus s')$, and
- the superscripts represent the power operator, here applied to all pixels in $(a \oplus s)$ and $(a \oplus s')$.

Without going into too much detail at this moment on exactly how they work, ***s*** and ***s'*** are matrices that are defined as:

$$s = \begin{bmatrix} -1 & \phi & 1 \\ -2 & \Diamond & 2 \\ -1 & \phi & 1 \end{bmatrix} \quad \text{and} \quad s' = \begin{bmatrix} -1 & -2 & -1 \\ \phi & \Diamond & \phi \\ 1 & 2 & 1 \end{bmatrix}$$

where the diamond centers the transform on the particular pixel of the image ***a*** that is currently being operated upon and the nulls indicate that the corresponding pixels in the image do not participate in this operation. The constants are weights to be multiplied with the pixels to which they correspond (for each specific placement of the diamond on a pixel of ***a***). For each centering of a template's diamond cell on a pixel of the image ***a***, the operator $\oplus$ defines the formula for computing the pixel corresponding to the diamond cell in the resulting image. More specifically, the operator $\oplus$ defines a summation of the products of each weight in the template matrix times its corresponding pixel in the image ***a***. Different Image Algebra operators will induce different pixel level arithmetic formulas. Finally, without loss of generality, the example is simplified by the assumption that each pixel has a single gray scale channel of unspecified precision.

3.2 AO processing of the example

For the moment, we will ignore the first two stages of AO, which just manipulate the expression to maximize the optimization opportunities. The reduction process starts at the top of the expression tree looking for patterns that will trigger transformation rules and recursively descends the tree until it finds a subtree that will trigger a transformation. No transformation can fire until the reduction engine gets to the image ***a*** within the subtree $(a \oplus s)$, at which point the transformation **CompositeLeaf**[6] causes the creation of a translator generated variable ***p*** of type pixel with an attribute adornment of the form $\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel))$[7] that replaces ***a*** in the subexpression, producing:

$$b = \begin{bmatrix} ((p\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)) \oplus s))^2 \\ + \\ (a \oplus s')^2 \end{bmatrix}^{1/2}$$

The adornment $\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel))$ anticipates how ***p*** will compile and the relationship of the compiled form of ***p*** to the iteration over ***a***. That is, ***p*** will compile into expressions of the form ***a[i,j]*** where ***i*** and ***j*** are of type int and index the image ***a***. The type of ***a[i,j]*** is pixel. Further, ***i*** and ***j*** will be the loop iteration variables and will range over all pixels in ***a***. This quantifier adornment is just shorthand for the essential information about the anticipated compiled form. Importantly, by being expressed as an attribute of ***p***, it avoids altering the AST structure of the expression which reduces the case complexity of the transforms. The adornment has the advantage that it avoids premature commitment of the abstract operator expressions to implementation level code, it lends itself to the manipulations that are required by the migration and merging of the quantifier adornments, and it allows the translation system to change course if later global interactions reveal greater opportunities for optimizations.

No further transformation progress can be made on the leaf ***p***, and the transformation engine moves up the tree to the expression $(p \oplus s)$ whereupon the transformation **BackwardConvolutionOnLeaves** fires, moving the attribute up to the expression level.

$$b = \begin{bmatrix} ((p \oplus s)\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)))^2 \\ + \\ (a \oplus s')^2 \end{bmatrix}^{1/2}$$

The $(a \oplus s')$ operation will be transformed

---

[4] This is one of a small number of basic operators that are claimed to perform all necessary image operations.

[5] In the IA notation, plus is overloaded to allow expressing the addition of images. The compilation process will reduce this addition to the addition of integer pixel values.

[6] Transformation names are included to relate to later architectural descriptions.

[7] I will use the publication form *_AttrLabel(AttrExpression)* to represent attribute adornments of expressions. In the examples in this paper, I will use only one attribute label *_Q* for "quantifier." Such adornments are not part of the domain specific expression per se but are associated translation state information that will be manipulated by the set of transformations operating on the target expression.

analogously producing the AST subtree:

$$b = \begin{bmatrix} ((p \oplus s)\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)))^2 \\ + \\ ((q \oplus s')\_Q(\forall q{:}(a[k{:}\text{int}, l{:}\text{int}]{:}\, pixel)))^2 \end{bmatrix}^{1/2}$$

The power of 2 operator belongs to a class of operator that has an associated transform (i.e., **ArithOpsOnComposites**) that recognizes the quantifier adornment on its operand and simply migrates the adornment to its level in the AST. This produces a tree that looks like:

$$b = \begin{bmatrix} (p \oplus s)^2\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)) \\ + \\ (q \oplus s')^2\_Q(\forall q{:}(a[k{:}\text{int}, l{:}\text{int}]{:}\, pixel)) \end{bmatrix}^{1/2}$$

The + operator belongs to a class of operator with an associated transformation (i.e., **ParallelOpsComposite-Operands**) that can handle the special case condition that the image operands iterated over by ***p*** and ***q*** are identical (i.e., both iterate over ***a***). This transformation merges the quantifiers and migrates the results to adorn the + operation. So, the AST now looks like:

$$b = \begin{bmatrix} ((p \oplus s)^2 + (p \oplus s')^2) \\ \_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)) \end{bmatrix}^{1/2}$$

The square root operator belongs to the same operator group as the square operator so this triggers the migration of the quantifier adornment to the square root operation (again using the **ArithOpsOnComposites** transform).

$$b = \begin{bmatrix} \left[(p \oplus s)^2 + (p \oplus s')^2\right]^{1/2} \\ \_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)) \end{bmatrix}$$

Next, the AO algorithm encounters the leaf composite ***b***[8] and adorns it again using the transformation **CompositeLeaf**.

$$\begin{bmatrix} c\_Q(\forall c{:}(b[r{:}\text{int}, s{:}\text{int}]{:}\, pixel)) \\ = \\ \left[\begin{bmatrix} (p \oplus s)^2 + (p \oplus s')^2 \end{bmatrix}^{1/2} \right. \\ \left. \_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel)) \right] \end{bmatrix}$$

Then, the assignment operator triggers the transformation **ParallelOpsCompositeOperands** to merge and migrate the quantifier adornments on the left and right hand sides of the assignment. In this case, the left and right operands are different (i.e., the pixel ***c*** in ***b*** and pixel ***p*** in ***a***), so the attribute adornment must express the idea that while the abstractions are distinct, the iteration and iteration variables ***i*** and ***j*** over their containers (i.e., ***b*** and ***a***) can be shared[9].

$$\begin{bmatrix} c = \left[(p \oplus s)^2 + (p \oplus s')^2\right]^{1/2} \end{bmatrix} \\ \_Q(\forall \langle c | p \rangle{:}(\langle b | a \rangle[i{:}\text{int}, j{:}\text{int}]{:}\, pixel))$$

The $\langle item1 | item2 \rangle$ groupings in the adornment express the idea that the composites must be processed in parallel. The shared items such as the subscripting variables ***[i,j]*** are shown only once. At this point, the overall expression can be turned into code. The quantifier and the declarations of composites to which it refers (e.g., ***a***) supply all of the information necessary to produce the loop iteration code (including the declarations for the loop iteration variables ***i*** and ***j***). They also supply the necessary information to translate the various abstractions such as ***c*** and ***p*** into code. It tells the compiler that ***c*** and ***p*** are coordinated through the loop iteration variables ***i*** and ***j***.

From this adornment, **GenerateLoopStructure** will generate the form:

```
for (i=0; i < m; i++)
    for (j=0; j < n; j++)
        b[i,j] = sqrt(power((a[i,j] ⊕ s),2) +
                 power((a[i,j] ⊕ s'),2));
```

[8] Actually, we are describing this out of the actual execution order to simplify the expression forms we have to show in this example. In reality, ***b*** would be the first leaf encountered and adorned. That adornment would be inert through all of the steps described so far and would not be involved in a merge until both subtrees of the assignment operator (=) had been processed and the AO method re-visits the assignment operator for the last time (i.e., on the last return from processing subtrees). The level of complexity of this reality seems excessive and adds nothing to the example. Hence, I have reversed the order.

[9] Due to space limitations, I have made the simplifying assumption that the images ***b*** and ***a*** have the same dimensions. Such simplifications are not inherent to the general method.

This expression can be further reduced (via **GenerateCodeForPower, GenerateCodeForFunctions**, and **ReduceConvolution-Operator**) to

```
for (i=0; i < m; i++)
    for (j=0; j < n; j++)
        { t1 = gcon10(a,i,j,s);
          t2 = gcon (a,i,j,s');
          b[i,j] = sqrt(t1*t1 +  t2*t2)  }
```

The above form is the final result using the techniques and transformations discussed in this paper. However, in point of fact, other AO transformation techniques not discussed in this paper can further reduce this form to:

```
for (i=0; i < m; i++)
    { im1=i-1; ip1= i+1;
      for (j=0; j < n; j++)
          { if(i==0 || j==0 || i==m-1 || j==n-1)
              then b[i, j] = 0;
              else { jm1= j-1; jp1 = j+1;
                     t1 = a[im1, jm1] * (-1) + a[im1, j] * (-2) +
                          a[im1, jp1] * (-1) +  a[ip1, jm1] *1  +
                           a[ip1, j] * 2  +  a[ip1, jp1] * 1;
                   t2 = a[im1, jm1] * (-1) +  a[i, jm1] * (-2) +
                        a[ip1, jm1] * (-1) +   a[im1, jp1] *1  +
                        a[i, jp1] * 2  +  a[ip1, jp1] * 1;
                   b[i, j] = sqrt(t1*t1 + t2*t2 )}}}
```

Because of space limitations, we will not describe in this paper how this last optimization step is accomplished.

4. The AO method

4.1 Overview

The AO method is a multi-phase walk of the AST in which transformations are triggered at each subtree based on the pattern of the domain specific operators, operands, and attributes of the nodes within that subtree, or within its scope, or within declarations referenced from that subtree. These transformations modify the tree in various ways. They may map domain operators or operands into conceptually lower level domain operator or operands. They may create new variables for the generated code. They may adorn the subtree with attributes that anticipate how the operators and operands will be implemented. They may merge the adornments thereby anticipating optimizations in the target implementations. They may reorganize the domain specific code to foster improved optimizations in the generated implementations.

4.1.1 Domain abstraction hierarchy

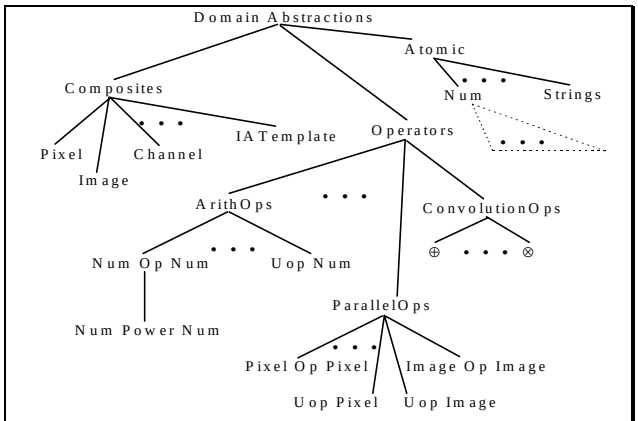

Figure 2
Domain Specific Abstraction

The domain specific operators and operands are organized into an abstraction hierarchy (see Figure 2), each node of which represents a set of AST patterns that will compile into target code with the similar data structure, flow and dependency patterns.

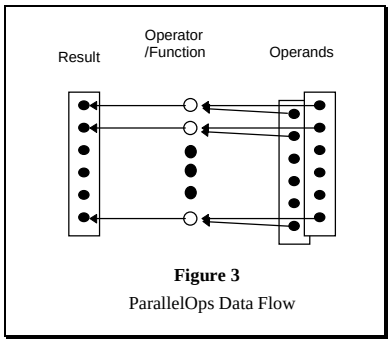

Figure 3
ParallelOps Data Flow

For example, instances of the class ParallelOps have the kind of data flow pattern shown in Figure 3 in which each result value within some composite structure (e.g., an array) is dependent upon corresponding values in the composite operands and upon no other values (except perhaps constants). Further, the operator or function is mathematically pure and does not produce any side effects.

The convolution operators, on the other hand, have a more complex dependency structure (shown in Figure 4) that anticipates an implementation consisting of an outer loop (or loops) within which each convolution window is computed by another nested loop.

[10] The function gcon is the general convolution function which uses ***s*** (or ***s'***) to compute the convolution value for the pixel ***a[i,j]***.

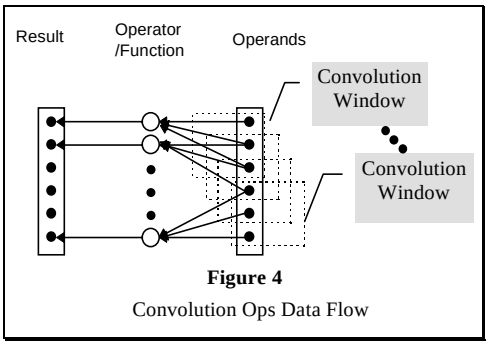

Figure 4
Convolution Ops Data Flow

Thus, the abstraction hierarchy provides:

1. An **organizational structure** in which to attach the transformations that will translate and optimize expressions of domain specific operands represented by the abstraction (e.g., the transformation **ParallelOps-CompositeOperands** used in the earlier example is attached to the "ParallelOps" abstraction),

2. An implied set of **data flow and dependency assumptions** (e.g., as shown in Figures 3 and 4) that will be true for all domain specific expressions that are instances of the abstraction represented by the node and therefore, that will be the assumptions built into all transformations that can potentially apply to those instances,

3. A **pattern** characterizing the AST structure of the current node (e.g., "Image ConvolutionOp IATemplate"), which specifies some but not all of the triggering conditions for applicable transformations, and

4. A framework for **transformation inheritance** (e.g., if all transformations attached to the node "Num Power Num" abstraction fail to trigger, then transformations attached to "Num ArithOp Num" will be tried, and so forth until some transformation triggers or until all transformations up that leg of the abstraction hierarchy have been tried and have failed to trigger).

4.1.2 Phased translation strategy

There is a second important organizing principle. The transformations are organized into distinct phases, each of which has a well defined but distinct translation purpose. The simplified AO process described here comprises four phases with the following translation purposes:

1. Inline any domain specific functions that according to programmer attached attributes (e.g., _inline attribute) will enhance optimization possibilities.

2. Move out of line subexpressions that will impede optimization (e.g., a convolution operator applied to an expression containing a convolution operator).

3. Successively reduce composite abstractions (e.g., images) to lower level composites (e.g., pixels) and eventually to programming language constructs (e.g., integers). Simultaneously, create attribute adornments that anticipate the extended computation required to operate on all elements of each composite (e.g., $\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel))$. Then migrate those adornments up the expression tree and merge them, thereby anticipating optimizations for those computations.

4. Generate optimized code from the combination of operators, operands, and their associated adornments.

As a consequence of this organization, each transformation is phase specific and performs a relatively simple operation. Table 1 shows the relationships between the example transformations, the abstraction hierarchy, and the phases.

4.1.3 Modifiers within optimization tags

User-defined modifiers can be added to any quantifier tag or associated directly with portions of the target program. For example, the modifier *_PromoteAboveLoop(j, ConstantExprOf(i))* will convert arithmetic expressions of *i* and constants into assignments and move them just above and outside of the loop controlled by *j*. Modifiers serve as optimization hints (or more operationally, as invocations of deferred optimizing transformations). They are used during code generation to produce variations or optimizations in the final code. Modifiers anticipate optimizations that cannot be executed until later when the generated code actually exists. Such modifiers stage the optimization steps to be performed at a future point in code generation.

The modifiers express anticipated optimizations ***in abstract terms*** (i.e., without reference to the exact structural forms of the generated code). For example, *ConstantExprOf(i)* will match any arithmetic expression involving only *i* and constants. This simplifies the optimization case logic by reducing an infinite variety of forms that are conceptually equivalent but structurally varied to a small number of abstract cases.

Modifiers can be *event-driven* and thereby can be associated with user-defined optimization events (e.g., *SubstitutionOfMe*) whose occurrence will trigger the modifier's execution, e.g., *_On(SubstitutionOfMe, _Fold0).* The final code for the example shown earlier contains loops that have been unwrapped into formulas and index expressions that have been promoted through mixtures of simple and event-driven modifiers.

Attached To	Phase 0 Inline DS Functions	Phase 1 Remove Impediments	Phase 2 Anticipate Optimizations	Phase 3 Generate Code
Root	InLine	BreakNestedConvolutions		
Composites			CompositeLeaf	ReduceLeafAbstractions ReduceAbstractDeclarations
Operators				GenerateLoopStructure
ArithOps				GenerateCodeForFunctions
(Num Power Num)				GenerateCodeForPower
ParallelOps			ParallelOpsCompositeOperands ArithOpsOnComposites	
ConvolutionOps			BackwardConvolutionOnLeaves ForwardConvolutionOnLeaves	ReduceConvolutionOperator

Table 1: Overview of AO Method

4.2 Example Transformations

Let us examine two AO transformations to provide a sense of how they are structured and how they operate. The Phase 2 **BackwardConvolution-OnLeaves** transform recognizes expressions like $(p\_Q(\forall p{:}(a[i{:}\text{int}, j{:}\text{int}]{:}\, pixel))\oplus s)$ and moves the quantifier up to the convolution operator level in the AST. The pseudo-code is:

```
Define Transform BackwardConvolutionOnLeaves( ) _Phase(2)
{if (Pattern(`(($pix : Pixel _Q( $Quantifier)) © $template :
            IATemplate) & both $pix and $template are leaves)))
then { Move the _Q($Quantifier) adornment from the $pix
       operand to the ($pix © $template) expression;
       return(True)}
else return(False) }}
```
[11]

The first thing that the transform does is to call Pattern to check the structure and type information of the current subtree in the AST, and to bind parts of the expression subtree to transformation variables. The conditions for the transform to succeed are that the left operand, which is bound to $pix, is of type Pixel and the right operand, which is bound to $template, is of type IATemplate. Further, both must be leaves of the AST and the $pix operand must have a quantifier adornment, whose value is bound to $Quantifier. If all of these conditions are true, the transformation removes the adornment on $pix, re-attaches it at the expression level, and returns true indicating success. If any of these conditions fail, the transformation as a whole will fail and processing of the current node of the tree (if any) will have to be accomplished by some other transform from farther up the data abstraction hierarchy. While not all transforms are this simple, most are simple with a small amount of localized processing.

Now let's look at the Phase 3 (code generation) transformation that actually generates the loop prefix structure -- **GenerateLoopStructure**. In truth, this transformation is fairly large with several distinct cases (e.g., handling arrays of different dimensions). These extensions are beyond the space limits of this paper. Thus, I will present a simplified version that reveals the essential character without the excessive details.

The processing is pretty straightforward. By the time AO method Phase 3 starts, the quantifiers will have migrated up the expression tree as far as they can go and will have merged with other applicable quantifiers[12]. Thus, this transform just detects expressions ($Expression) with quantifiers on them, replaces all abstract references from the $Abs list (e.g., the pixel ***p*** from the example) with corresponding implementation references from the $CompositeDS list (e.g., ***a[i,j]*** from the example) and wraps the resulting expression with the appropriate loop prefix structure. $GetDimension is a generation-time function that fetches the dimensional information (constants or code) from the AST for use in constructing the loop prefix.

```
Define Transform GenerateLoopStructure( ) _Phase(3)
{if ( Pattern( `($Expression _Q( ∀$Abs : ($CompositeDS[$Idx1:
                int, $Idx2: int] : $CompositeDSType) )) ))
then { Generate declarations for $idx1 and $Idx2 and
       put them in the scope of current statement;
       Replace each $Absk reference in $Expression with
           `($CompositeDSk[$Idx1, $Idx2]), where $Absk and
           $CompositeDSk are the corresponding elements on
           the $Abs and $CompositeDS lists;
       Replace current subtree with
```

[11] The symbol © represents the class of all convolution operators.

[12] Variations induced by composites that do not perfectly correspond can be handled by AO but are beyond the scope of this paper.

```
        `{(for($Idx1=0;
            $Idx1 < ($GetDimension($CompositeDS, 1) - 1) ;
            ++ $Idx1)
             {(for($Idx2=0;
                $Idx2 < ($GetDimension($CompositeDS, 2)-1) ;
                ++ $Idx2)
                 $Expression }}
      Return(True); }
. . . Code for other cases elided . . .
 else return(False); }
```

5. Related Research

5.1 Generation systems

The Draco program generation system shares the anticipatory, domain oriented optimization philosophy of AO, but its published descriptions do not mention any AO-like optimization methods for melding portions of the code (e.g., loop prefixes). (See Neighbors 1980, 1983, and 1989.) Typically, Draco optimizations are small-grained, forward refinement-based rewrites of expressions such as rewriting "power(x,2)" as "x*x".

Aspect Oriented Programming (AOP) shares with AO the intention and the accomplishment of reweaving code for efficiency. (See Kiczales, *et al.*, 1997.) The main difference appears to be that AOP performs its code reweaving in a mixture of problem domain and implementation level constructs (e.g., the AOP programmer can write explicit looping constructs). By contrast, AO prevents the programmer from writing implementation level code (e.g., explicit looping constructs) by hiding the implementation form. He cannot know what the implementation structures look like because they are indeterminate until the generation-time moment at which they are actually created. By this tactic, AO avoids open-ended searches (e.g., data flow analysis) that are difficult to avoid in a mixture of domain level and implementation level constructs. In AO, implementation level inference is handle via the annotation tags, which are logically separated from the program code. They serve as a design blackboard where automated program reweaving strategies can be mapped out and revised without altering the structure of the program code until the moment when the final rewoven implementation code is generated as whole cloth.

Other generation methods address complementary aspects of the problem. See Batory, Novak, and Smith.

5.2 Conventional optimization techniques

Each of the techniques that I have discussed in the AO method have their analogs in the conventional programming language context. (See Bacon *et al.*, 1994.) with the fundamental difference being that AO achieves the optimizations directly, using the data flow and dependency information implied by the domain specific operators and operands (e.g., addition on images) and never creates the non-optimized form of the code. Conventional optimization, on the other hand, starts with the non-optimal (i.e., "compiled") code in a conventional programming language and then recovers the flow and dependency information -- information that is more directly available from the domain specific operators and operands -- via complex analysis techniques that induce large, open-ended searches. Further, conventional optimization often misses opportunities for optimizations because of the difficulty of the inference problems (e.g., variable alias analysis). AO translation rules are specifically designed to prevent aliasing problems that would inhibit optimizations. In contrast, the freedom of a general programming language provides no such discipline and therefore, it is this very freedom that is the source of many difficult optimization problems.

AO avoids such difficulties by recasting the problem into a form handled by simpler and more effective methods.

5.3 Optimization in APL, FORTRAN, and parallel processing languages

While optimization in APL, FORTRAN, and parallel processing contexts can often achieve a subset of the optimization results of AO, the methods are fundamentally different, sometimes complementary, and generally more restrictive than AO optimization. Representative of the techniques that optimize looping over expressions of high level operations on arrays and matrices are the APL techniques based on the general idea of "steppers." (See Guibas and Wyatt, 1978 and Ju *et al* , 1992.)

Stepper-like optimization methods allow the direct fusing of loop iteration prefixes for loops whose ranges and increments match exactly, or direct fusing of subloops that are decomposed specifically to achieve the matches. A stepper is basically a characterization of loop iteration behavior in terms of the array axis, starting index, stride (i.e., increment), and shape of target array. Operationally, a generic stepper expression is attached at the top of an APL expression tree. It is propagated down the tree and transformed by operator specific transformations for each operator that it passes through on the way to the expression's leaves. The resulting steppers at the leaves contain the data access pattern of the associated source array that is required to compute an element of the target array in the expression. These patterns are then used to generate the set of common loop prefixes and the accessor expressions for each of the arrays used in the expression.

The main difference between such APL optimization

methods and the AO method is the operational context, with AO operating in the domain context and APL operating in the programming language context. Thus, any optimization opportunity that depends on the use of domain specific knowledge (e.g., the data flow implied by convolution) or user-defined knowledge is beyond the built-in APL techniques. The example seen earlier could not be handled directly by APL optimization techniques. Nevertheless, these techniques are complementary to AO and could be easily incorporated through transformations thereby providing the ability to deal with matrix domain abstractions that have different shapes.

6. Conclusions

There are several key innovations in AO. 1) Annotation tags that separate the optimization planning from the actual code structure and thereby reduce the variations in control and data forms that the transformations must deal with; 2) modifiers that capture anticipated optimization transformations that must be deferred until the code to be optimized actually exists; 3) inference rules (i.e., transformations) that operate strictly in the domain of the problem oriented operators and operand types, 4) modifiers expressed in abstract terms such that each modifier expression applies to an infinite number of variations of the concrete implementation code thereby simplifying AO's case logic; 5) meta-optimization transformations that reconcile, make consistent and order sets of interdependent modifiers and thereby allow reasoning about the optimization process itself before it is actually executed; 6) the encoding of implicit data flow and data dependency relationships in the implementations of the domain specific operators and operands via the transformation inheritance hierarchy and the transformations that are attached to nodes of that hierarchy; and 7) event-driven modifiers that allow interdependent, anticipated optimizations to be organized on a time line that assures their consistent application.

AO is an improvement over conventional optimization and generation techniques because it eliminates the open-end searches (e.g., in data flow, dependency and alias analysis) required by conventional optimization strategies as well as the open-ended rule searches required by large, flat corpora of optimization rules. It recasts the optimization problem into a form that is handled by much simpler and more effective methods. AO opens up opportunities for true domain specific languages without the burden of poor performance of the compiled code or poor performance of the generator because of open-ended searches.

7. Acknowledgements

I would like to thank my graphics domain expert Brian Guenter for introducing me to the Image Algebra and teaching me enough about imaging to be dangerous.

8. References

David F. Bacon, Susan L. Graham, and Oliver J. Sharp, *Compiler Transformations for High-Performance Computing*, **ACM Surveys**, Vol. 26, No. 4, December, 1994.

Don Batory, and Sean O'Malley, *The Design and Implementation of Hierarchical Software Systems*, **ACM Transactions on Software Engineering and Methodology**. Vol. 1, No. 4, pp. 355-39, October, 1992b.

Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, *Scalable Software Libraries*, **Symposium on the Foundations of Software Engineering**. Los Angeles, CA, December, 1993.

Ted J. Biggerstaff, *Directions in Software Development and Maintenance*. Keynote Address, **Conference on Software Maintenance, Montreal**, Canada, October, 1993a.

Ted J. Biggerstaff, *The Library Scaling Problem and The Limits of Concrete Component Reuse*, **International Conference on Software Reuse**, November, 1994.

L. J. Guibas and D. K. Wyatt, *Compilation and Delayed Evaluation in APL*, Fifth ACM **Symposium Principles of Programming Languages**, pp. 1-8, 1978.

Gwan-Hwan Hwang; Jenq Kuen Lee; Dz-Ching Ju, *An array operation synthesis scheme to optimize Fortran 90 programs*, Conference Title: **Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)**, July, 1995; Published in SIGPLAN Notices, Vol. 30, No. 8, pp. 112-22, August, 1995

Dz-Ching Ju, Chuan-Lin Wu, and Paul Carini, *The Synthesis of Array Functions and its Use in Parallel Computation*, **Proc. Int'l Conference on Parallel Processing**, pp. 293-296, 1992.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maede, Cristina Lopes, Jean-Marc Loingtier and John Irwin, *Aspect Oriented Programming*, **Tech. Report SPL97-08 P9710042**, Xerox PARC, 1997.

James M. Neighbors, *Software Construction Using Components*, **Ph.D. Dissertation**, Univ. of Calif. at Irvine, 1980.

James M. Neighbors, *The Draco Approach to Constructing Software from Reusable Components*, **Workshop on Reusability in Programming**, Newport, RI , 1983.

James M. Neighbors, Draco: A Method for Engineering Reusable Software Systems. In Ted J. Biggerstaff and Alan Perlis (Eds.), **Software Reusability**, Addison-Wesley/ACM Press, 1989.

Gordon S. Novak, *Creation of Views for Reuse of Software with Different Data Representations*, **IEEE Transactions on Software Engineering**, Vol. 21, No.12, December, 1995.

G. X. Ritter, J. N Wilson, and J. L. Davidson, *Image Algebra: An Overview*, **Computer Vision, Graphics, and Image Processing,** 49, pp. 271-331, 1990.

G. X. Ritter, **Image Algebra**, University of Florida, 1993.

G. X. Ritter and J. N Wilson, **Handbook of Computer Vision Algorithms in Image Algebra**, CRC Press, 1996.

Douglas R. Smith, *KIDS—A Knowledge-Based Software Development System*, in **Automating Software Design**, M. Lowry & R. McCartney, Eds., AAAI/MIT Press, 1991.