# Composite Folding and Optimization in Domain Specific Translation

Ted J. Biggerstaff

June, 1998

Technical Report
MSR-TR-98-22


Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# Composite Folding and Optimization in Domain Specific Translation

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

## Abstract

*Combinatorial increases in programming productivity are possible by the introduction of high level operators and operands for composite data structures such as arrays, matrices, trees, record composites, etc. Such operators and operands can be composed to generate an infinite variety of* ***virtual*** *reusable components. However, the performance of the code resulting from such compositions is often inadequate because of the code fragmentation and delocalization engendered by the building blocks. Attempts to automatically reorganize this code to optimize performance using conventional optimization strategies are usually computationally impractical because of the large search spaces engendered.* ***Anticipatory Optimization*** *(**AO**) is a method for compiling such compositions directly to optimized code without large search spaces and indeed, without any search space at all. The keys to AO are 1) distributed optimization plans expressed via code annotations and 2) transformation based optimization planning that concurrently reasons over the domain information, the generated program and the evolving optimization plan .*

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

## 1. The problem

From a generator builder's point of view, domain analysis is the process of identifying domain specific operators and operands plus the transformations that can *compile* these operators and operands. The choice of how to define such operators and operands leads to a quandary between choosing operators that exhibit powerful programming amplification through the combinatorial opportunities provided by composition, and choosing operators and operands that can be easily transformed into high performance code. The central difficulty revolves around delocalization of information. [10] If I factor the operators and operands into highly general constructs, I can write combinatorially many compact expressions with them that effectively form an infinite *virtual library* of reusable components, one component for each possible composite expression. However, this means that the tightly integrated information needed by the compiler to generate high performance code is split across many operators, operands and subexpressions. With current technology, compiling such expressions requires huge search spaces of possible transformation sequences to assure finding the optimal localizations for high performance execution.

On the other hand, if I define less general operators and operands in which cross operator and cross expression code is already localized for performance reasons, the number of possible variations that can be produced by my generator drops precipitously and my infinite virtual library very likely becomes a finite library. [3-6]

An example of this delocalization is illustrated by the Image Algebra [13] expression for Sobel edge detection in bitmapped images [ 4-6].

***Expression 1:***

**image a, b :form ( array m n);**
**b = [ (a ⊕ s)² + (a ⊕ sp)²]^1/2 ;**

where **a** and **b** are **(m X n)** grayscale images, ⊕ is a convolution operator that applies the Image Algebra (IA) template matrices **s** and **sp** to each pixel **a[i,j]** and its surrounding neighborhood in the image **a** to compute the corresponding pixel **b[i,j]** of **b**. **s** and **sp** are reusable OO components that define the specifics of the pixel neighborhood. In the domain specific expression, the (implied) looping is delocalized. The subexpressions **(a ⊕ s), (a ⊕ s)$^2$ , (a ⊕ sp) , (a ⊕ sp)$^2$ , [ (a ⊕ s)$^2$ + (a ⊕ sp)$^2$ ], [ (a ⊕ s)$^2$ + (a ⊕ sp)2]$^{1/2}$,** and **b = [ (a ⊕ s)$^2$ + (a ⊕ sp)$^2$]$^{1/2}$** imply six or seven distinct loops. In the ideal executable code for a conventional sequential machine, all of these loops would be optimally encoded as one loop within which all of the operations would be accomplished. Further, expressions of the form **(a ⊕ s)** delocalize the information of the convolution. The ⊕ operator provides the *general* convolution formula (i.e., the sum of the products of the pixel values times the weights associated with the pixel's position within some pixel neighborhood) and the methods of the IA templates **s** and **sp** provide the

 

*specific* coordinates of the neighboring pixels, the specific weights associated with each neighboring pixel, the dimensions of the neighborhood, and any special case behavior (e.g., image boundary processing). In the ideal executable code, the general formula for ⊕ and specific details of **s** and **sp** would be tightly integrated. However, achieving this via automatic localization is made difficult by the fact that the various values supplied by **s** and **sp** may be arbitrarily complex computations, e.g., conditional and/or parameterized expressions. For example, the semantics of the ⊕ operator are defined as

**(a ⊕ s) ⇒ {∀i,j :{Σk,l : {a[row.s(a[i,j],k,l), col.s(a[i,j],k,l)]*w.s(a[i,j],k,l) }}}**

where **row.s(a[i,j],k,l)**, **col.s(a[i,j],k,l)**, and **w.s(a[i,j],k,l)** are methods of **s** that define the specific **a[i,j]** neighborhood coordinates and associated weights. In this example, **a** has dimensions **(m X n)** and the ranges of **k** and **l** are defined for both **s** and **sp** as **[-1:1]**. The problem is that the definitions of the methods of **s** may contain arbitrarily complex expressions that likely will need to be reformulated, moved, and combined across loop, statement, and expression boundaries to achieve optimal code localization for performance. For example, the definition of a weight -- **w.s(a[i,j],k,l)** -- depends on whether the neighborhood defined by **s** is completely within the image **a** or a portion of that neighborhood is hanging over the edge of **a**. Thus, for **s** and **sp** defined (for expository purposes) by the matrices

$$s = \begin{bmatrix} -1 & \phi & 1 \\ -2 & \Diamond & 2 \\ -1 & \phi & 1 \end{bmatrix} \qquad sp = \begin{bmatrix} -1 & -2 & -1 \\ \phi & \Diamond & \phi \\ 1 & 2 & 1 \end{bmatrix}$$

with the diamond centered on some **a[i,j]** focus pixel and ϕ representing non-participating pixel locations, then **w.s** is defined as

```
w.s(a[fi,fj],fp,fq) ⇒
{if (( fi == 0) || ( fj == 0) (fi == ( fm - 1)) || (fj == (fn - 1)))
    then 0;
    else {if (( fp != 0) && ( fq != 0)) then fq;
              else {if ((= fp 0) && (fq != 0)) then (2 * fq);
                        else 0 }}}
```

and the **row** and **col** methods are defined as **row.s(a[fi,fj],fp,fq) ⇒( fi + fp)** and **col.s(a[fi,fj],fp,fq)) ⇒( fj + fq)**. One might be inclined to simply inline these definitions for **w**, **row**, and **col**, and then partially evaluate them to simplify. But this alone yields pretty inefficient code – seven loops, five temporary images, and many redundant expressions. In fact, the *localized* code we would like to produce is

```
for (i=0; i < m; i++)                          /*Expression 2*/
    { 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 )}}}
```

This result can only be achieved if a long series of very specific transformations are executed in a very specific order. But this requirement engenders a very large search space of optimizations. For example, if-then statements are merged, loops are unwrapped, constants are folded, expressions are moved into and outside of loops, expressions are simplified via partial evaluation, and common subexpressions are eliminated. It would be computationally impractical for a general optimization system to find this optimal solution in the huge space of possible final forms for this code.

## 2. A Solution

A solution to this problem (called *Anticipatory Optimization*) [4-6] anticipates an abstract optimization plan and expresses this plan as a set of tags affixed to various reusable components. These tags are effectively deferred invocations of optimizing transformations. Thus, the transformations are distributed in space (i.e., over the program's components) and in time (e.g., their triggering times are expressed as optimization event expressions in the tags). Domain specific expressions (e.g., the expression for Sobel edge detection) to which tags be attached, serve as a design blackboard where automated program optimization and localization strategies can be mapped out (i.e., anticipated), reasoned about, and revised without altering the structure of the domain specific expression until the moment when the final rewoven implementation code is generated as whole cloth. The transformations that implement this optimization strategy use three kinds of information simultaneously: 1) program language semantics; 2) domain knowledge; and 3) optimization planning knowledge (i.e., the tags) to perform this inference process, thereby allowing the transformations to reason about the optimization process as well as the program. This differentiates AO from other translation strategies that only reason about the program. This strategy reduces the optimization search space, making each transformation choice a single choice. The choice is

predetermined to be exactly the right transformation for a given optimization event. The tags accomplish the reduction of the search space by identifying: 1) the specific transformation to be invoked (i.e., the one named in the tag), 2) the specific object on which it will act (i.e., the one to which it is attached), and 3) the specific time of invocation (by virtue of its location on the program or by virtue of an event expression defined in the tag).

## 3. An Example

### 3.1 Basic Definitions

Let us follow through the compilation of the Sobel edge detection expression shown in expression 1. While expression 1 is shown in a prettified publication form, the actual form for expression 1 is not quite as pretty because of its Lisp heritage (***Expression 3***):

```
(DSDeclare Image a :form (array m n) :of integer)
(DSDeclare Image b :form (array m n) :of integer)
(DSExpr (setf  b (sqrt (+ (expt (bconv  a  s)  2)
                          (expt (bconv  a  sp) 2)))))
```

In this expression, **bconv** is the convolution operator (⊕ in pretty form), **expt** is the exponentiation operator, **DSDeclare** is a Lisp macro that creates an instance of a CLOS type (in this case, an **Image** type), and **DSExpr** is a Lisp macro that creates an instance of an expression that is to be compiled using AO techniques.

Expression 3 is what the end user writes but of course this code draws upon components from a reusable library that define **s**, **sp**, **bconv**, and so forth. The reusable definitions for the reusable components are given in expression 4. ***Expression 4a***:

```
(DSDeclare IATemplate s
        :form (array (-1 1) (-1 1)) :of integer)
(DSDeclare IATemplate sp
        :form (array (-1 1) (-1 1)) :of integer)
```

where IATemplate is the Image Algebra template type and the methods of **s** are defined as follows. (**sp**'s methods are analogously defined.) ***Expression 4b:***

```
(Defcomponent PRange (s ?plow ?phigh)
   (_Range ?plow ?phigh))

(Defcomponent QRange (s ?qlow ?qhigh)
   (_Range ?qlow ?qhigh))

(Defcomponent Row (s  #.ArrayReference ?p ?q)
  (+ ?iter1 ?piter))

(Defcomponent Col (s  #.ArrayReference ?p ?q)
  (+ ?iter2 ?qiter))

(Defcomponent W (s #.ArrayReference ?p ?q)
  (if (or (== ?iter1  ?i1low) (== ?iter2  ?i2low)
          (== ?iter1 ?i1high) (== ?iter2 ?i2high))
    (then 0)
    (else (if (and (/= ?piter 0) (/= ?qiter 0))
                  (then ?qiter)
              (else (if (and (== ?piter 0)   (/= ?qiter 0))
                             (then (* 2 ?qiter))
                             (else 0)))))))

(Defcomponent pixelXtemplate
  (BackwardConvolutionOp
    #.ImagePixelReference #.TemplateReference)
      (_Sum (?p ?q)
        (_SuchThat
          ( _Member ?p (PRange ?template ?plow ?phigh))
          (_Member ?q (QRange ?template ?qlow ?qhigh)))
           (* (aref ?aname
                 (row ?template
                    (aref ?aname ?iter1 ?iter2) ?p ?q)
                 (col ?template
                    (aref ?aname ?iter1 ?iter2) ?p ?q))
              (w ?template (aref ?aname ?iter1 ?iter2) ?p ?q)))
```

In these definitions, the forms "**?***varname*" are pattern variables (i.e., generation-time variables) used by the AO transformation system to hold portions of the evolving program. The formal parameter sequences of these method definitions may include so-called *enabling condition* patterns that are often included via the "#." Lisp reader macro, (e.g., **ArrayReference**). These are reusable *patterns* that in addition to providing a name for the parameter, verify expected relationships between the parameter and other domain specific entities (e.g., between **a** and its iterators **i** and **j**). Enabling condition patterns also typically check types and bind additional pattern variables needed by the definitions. For example, **PRange** binds **?plow** and **?phigh** through a pattern (i.e., **IATemplate-instance**) that matches **s**, then finds **s**'s P range iterator and then binds the iterator's low and high values of it to **?plow** and **?phigh**.

The definitions of expression 4b rely on the following pattern definitions. (***Expression 5)***:

```
(setf ArrayReference
  '$(por
    $(pand  (aref  ?aname #.Iter1AndRange  #.Iter2AndRange)
                   $(bindconst ?acase 2D))
    $(pand (aref  ?aname  #.Iter1AndRange)
                   $(bindconst ?acase 1D))))

(setf Iter1AndRange
          '$(pand ?iter1
                   $(psuch formals ?iter1
```

```
                    (_Range ?i1low ?i1high))))
(setf Iter2AndRange
      '$(pand ?iter2
              $(psuch formals ?iter2
                      (_Range ?i2low ?i2high))))

(setf BackwardConvolutionOp-instance '?op)

(setf ImagePixelReference
      '$(pand #.ArrayReference
              $(plisp (IsDSType ?aname 'Image))))

(setf TemplateReference
      '$(pand ?template
          $(plisp (IsDSType ?template 'IATemplate))))
```

These patterns will recognize, among other things, a one or two dimensional array reference of the form **(aref a i)**[1] or **(aref a i j)** binding the array name (e.g., **a**) to **?aname**, the array's iterator or iterators (e.g., **i** and/or **j**) to **?iter1** and **?iter2**, the upper and lower end of each iterator's range to **?i1low**, **?i1high**, **?i2low**, and **?i2high** respectively, and a tag (e.g., **2D**) to **?acase** to indicate the specific form recognized.

In addition, Defcomponent replaces the instance specifier **s** in a method's formal parameter list with a reusable pattern common to all instances of the class. This pattern expresses the enabling conditions required by the methods of **s**. For the class IATemplate, this pattern has the form (***Expression 6)***:

```
(setf IATemplate-instance
    '$(pand ?self
            $(psuch dimensions ?self
                 ((_Member ?piter
                           (_Range ?plow ?phigh))
                  (_Member ?qiter
                           (_Range ?qlow ?qhigh))))))
```

which binds the particular instance of the IATemplate to **?self** , binds the iterators generated for use by the instance to **?piter** and **?qiter**, and finally, binds the low and high values of the ranges of those iterators. All of these bindings are now available for use by the method bodies. The other patterns, viz. **ImagePixelReference** and **TemplateReference**, perform similar checking and binding services. Enabling conditions extend the notion of a formal parameter list by recognizing that domain specific design elements (e.g., a template or a pixel) are part of an interrelated fabric of design information that describes individual design elements and their inter-dependencies and that is needed by the method definitions.

The pattern matching engine is central to all of the operations of the AO generator (e.g., the viewer, partial evaluator, and the transformations). Pattern matching is performed by a unification-based, fully backtracking pattern matcher/inference engine with a user extensible array of operators that include: an "and" operator requiring that all of its patterns match the current data element (i.e., **$(pand** *pat1 pat2* …**)** ); an "or" operator requiring that at least one of its patterns match (i.e., **$(por** *pat1 pat2* …**)** ); an operator to match the values of CLOS slots (i.e., **$(psuch** *CLOSInstance Slotname Pattern***)** ); an operator to execute arbitrary Lisp code that can return bindings as well as succeed or fail (i.e., **$(plisp** *Lisp code* … **)** ); ?variables, which match and bind the current data item (e.g., **?iter1**); an operator that binds a variable to the data matched by an arbitrary pattern (e.g., **$(bindvar** *var Pattern***)**); an operator to bind a constant value to a variable (e.g., **$(bindconst ?acase 2D)** ), which is often used to simplify later case logic; an operator to span over uninteresting data items until it finds data of particular interest, after which the intervening values are bound to a variable (i.e., **$(spanto** *Var Pattern***)**); an operator to bind the remainder of a list (i.e., **$(remain** *Var***)** ; an operator to perform pattern matching on different data (i.e., **$(pmatch** *Pattern Data***)***;* an operator to perform a Prolog like cut, thereby abandoning any remaining choices at the last choice point (i.e., **$(pcut)** ); an operator to recursively perform matching using a pattern bound to a variable (i.e., **$(pat** *Var***)**); and a fail operator to cause the matching to backtrack to the last choice point (i.e., **$(pfail)** ) and restart the matching with the next choice. In addition, each transformation used by the AO generator is written as a pattern that matches and rewrites the expression tree. All pattern matching and inference within the AO generator is accomplished with this engine. While biased toward structure-based pattern operations the engine has the capability to perform Prolog like inferencing.

The following trivial example illustrates the backtracking behavior of pattern matching given failure. The first argument to the **with-matching** macro is the pattern, the second is the data, the third is the initial bindings (in this case, nil) and the remaining items are Lisp expressions that are executed upon completion of the pattern match. These expressions are wrapped with a Lisp *Let* containing one Let variable bound to the final binding of each ?variable. In this example, the Lisp Let variable **x** will be bound to the final binding of **?x,** in this case, nil.

[1] (aref a i) is an internal form. The AO generator provides a viewer that casts internal forms into a more intuitive publication form. (aref a i j) is cast into a[i,j], prefix expressions are converted to infix, variables that are internally stored as (*var* x) are cast into ?x and so forth. In the example in the succeeding section, we will express the evolving example in the publication form produced by the viewer.

```
(with-matching
    '( $(spanto ?x g)
       $(plisp (print ?x) ( terpri) t) $( pfail))
     '(a b c g d e f g h i j g g) nil
      (print x)  (print 'DONE))
```

This expression will print:
(A B C)
(A B C G D E F)
(A B C G D E F G H I J)
(A B C G D E F G H I J G)
nil
DONE

# 4. Compiling Domain Specific Expressions

## 4.1 Loop Fusion

The AO method is a multi-phase transformation process. The first phase performs loop fusion on expression 3 resulting in the following code.

```
for (i=0; i < m; i++)                          /*Expression 7*/
    for (j=0; j < n; j++)
        { t1 = bconv( a[i,j], s);
          t2 = bconv(a[i,j] sp);
          b[i,j] = sqrt(t1*t1 +  t2*t2)  }
```

Loop fusion is a multi-phase algorithm that walks the AST (Abstract Syntax Tree) triggering transformations at each subtree based on the pattern of the domain specific operators, operands, and attributes on the nodes. These transformations modify the tree in various ways. They may map domain operators or operands into conceptually lower level domain operator or operands (e.g., Image type **a** gets mapped into a Pixel **p** and finally into an Integer **a[i,j]**). They may create new variables for the generated code (e.g., **t1** and **t2**). They may adorn the subtree with attributes that anticipate how the operators and operands will be implemented (e.g., **(_Q (∀p:(a[i:Integer, j:Integer] :Pixel))** indicates that the pixel **p** will be implemented as an array reference **a[i, j]** where **i** and **j** are integers.). They may merge the adornments thereby fusing loops or for another point of view, rewriting the (anticipated) data flows implied by the domain specific operators and operands. Merging tags is one method by which AO anticipates optimizations in the target implementations while deferring their actual execution. Finally, they may reorganize the domain specific code to foster improved optimizations in the generated implementations.

The final pass of the loop fusion phase (the code generation pass) converts the adornments or tags into the appropriate loop forms and may perform various other optimizations particular to this stage of processing (e.g., the reduction in strength optimization of the forms **(expt (bconv a[i,j], s) 2)** and **(expt (bconv a[i,j], sp) 2)**. What is more, this transformation anticipates the possibility that at some future point in the optimization process, one or more instances of these expressions may simplify to a constant thereby, creating the opportunity for constant folding. The transformation embeds tags that anticipate these future possible optimizations, a service that will be critical later in the optimization process.

This anticipation is manifest as follows. When it introduced the **T1** and **T2** assignment statements, it introduced data flows into what had been a purely functional expression (i.e., the right hand side of the assignment). Future transformations will need to know about these data flows, so it adds tags that records the data flow dependencies. However, it also anticipates the possibility that one of the expressions may eventually reduce to a constant thereby providing the opportunity for constant folding within the **SQRT(T1 * T1 + T2 * T2)** expression. Hence, it adds the optimization tag

**(_ON (MigrationIntoBlockOf 'T1) (_SCHEDFOLDFLOWS))**

to the **T2** assignment and the tag

**(_ON (MigrationIntoBlockOf 'T2) (_SCHEDFOLDFLOWS))**

to the **T1** assignment. Whenever the **T1** or **T2** assignments get moved into the other's block, **_SCHEDFOLDFLOWS** will be awakened to check for the possibility of relocating the **SQRT(T1*T1 + T2*T2)** expression and performing constant folding. This opportunity will in fact occur in the example we will examine and cause the creation of the degenerate leg **b[i, j] = 0** seen in expression 2. We will come back to this subject later in the example.

The loop fusion phase is treated in greater detail in [4] and we will not consider it further in this paper. We will start with expression 7 and follow its evolution into the final code (expression 2).

## 4.2 Composite Folding

The next phase of processing, *composite folding*, inlines the methods of the various composite structures (e.g., IA templates) and executes the transformations in the tags distributed over the domain specific expression. In the course of this execution, certain optimization events (e.g., substitution of an expression or migration of an expression into a particular context) may trigger other so-called *event-driven* or opportunistic optimizations (e.g., transformations like zero folding, promotion of code above loops, moving context expressions into if-then-else forms, etc.). Optimization re-planning may be triggered along the way causing old optimization tags to be altered or new ones to be added.

Some optimization planning tags were added to the **i**, **j** loop during code generation phase of the loop fusion, notably

**(_CF (->bconv( a[i,j], s)) (->bconv(a[i,j] sp)))**

**(_PromoteAboveLoop J (ConstantExpressionOf I))**
**(_PromoteToTopOfLoop J (ConstantExpressionOf J))**

where the notation "->" implies a direct reference to the convolution expressions within the AST. The first of these tags will trigger composite folding (**_CF**) of the two referenced expressions, which inlines method definitions, simplifies the resulting code, and finally, attempts to share code across the subexpressions. The latter two tags anticipate the opportunity to promote common index expressions of **i** (e.g., **(i - 1)**) above and outside the inner loop controlled by **j**, and expressions of **j** to the top of but inside the loop controlled by **j**. If the loops generated from the **bconv** expressions get unrolled, these promotion transforms will trigger and otherwise they will not. We will follow **_CF**'s operation in some detail starting with the optimizations of the subexpression **bconv( a[i,j], s)**.

Basically, the local optimization phase of **_CF** is a process of inlining the definitions of **bconv** and the methods of **s**, and executing any AO generated or pre-positioned optimization tags. The definition used for the **bconv** expression is the body of the PixelXTemplate method of the BackwardConvolutionOp class (see expression 4b), which the AO viewer will present in a modestly more readable form (***Expression 8*** )[2]

```
T1 =
 (_SUM (?P ?Q)
   (_SUCHTHAT
      (_MEMBER ?P (PRANGE S ?PLOW ?PHIGH))
       (_MEMBER ?Q (QRANGE S ?QLOW ?QHIGH)))
   (A[(ROW S A[I,J] ?P ?Q),(COL S A[I,J] ?P ?Q)]
     * (W S A[I,J] ?P ?Q)))
```

The domain expert who wrote this component and entered it into the reuse library anticipated the possibility that the loops over **?p** and **?q** might be profitably unrolled and therefore, attached two optimization tags to the loop:

```
(_ON CFWRAPUPEND
    (_UNWRAPIFSMALLCONSTANT ?P |?PHIGH - ?PLOW |)
(_ON CFWRAPUPEND
    (_UNWRAPIFSMALLCONSTANT ?Q |?QHIGH - ?QLOW |)
```

These are event-driven optimization tags that will only be triggered when **_CF** posts the **CFWRAPUPEND** event, which is the point just after any integration of subexpression code has been accomplished and just before **_CF** returns. This is the **_CF** phase in which clean up transformations perform their work. Except for the transformations that perform the promotion of the constant expressions of **i** and **j**, these will be the last transformations to fire for the example.

While not shown in Expression 4b, the definition of the **W** method of **S** also has some tags that anticipate possible optimizations. The "if" test in **W** identifies the special case where the template **s** (and analogously **sp**) are partly hanging off the edge of the image and it is dependent only upon the image parameters (i.e., **i**, **j**, **m** and **n**) but not the target program variables that will be bound to **?p** and **?q**. Thus, it need not be recalculated for each iteration of any loop over **?p** or **?q** and therefore, can be moved outside of any such loop. This is anticipated by the tag

```
(_ON SUBSTITUTIONOFME
    (_PROMOTECONDITIONABOVELOOP ?P ?Q))
```

which triggers anytime the if statement gets substituted into a new context. It will attempt to promote the condition outside of any loop of **?p** and **?q** by incorporating that loop into the then and else clauses.

A second tag attached to the if statement

```
(_ON CFWRAPUP (_MERGECOMMONCONDITION))
```

anticipates that this special case test might be common across subexpressions and therefore, could be shared. Since this opportunity will not occur until after local optimizations within the separate subexpressions have run their course, this optimization it not triggered until **_CF** posts the event **CFWRAPUP**, which is the point in time **_CF** allows code sharing optimizations across subexpressions.

If the if statement does indeed get promoted outside of any loops controlled by **?p** and **?q**, the component developer anticipates that the "then" and "else" clauses will get substituted into some new expression contexts, providing additional optimization opportunities. For example, substituting zero typically provides an opportunity for simplification. Hence, the developer attaches to the zero on the then leg the tag

```
(_ON SUBSTITUTIONOFME (_FOLD0))
```

If and when this transformation gets triggered, it will eat up the expression tree eventually simplifying away everything up to and including the loop in which it occurs. This transformation is key to formation of the **b[i,j] = 0** in expression 2.

Similarly, the substitution of the if-then-else expression on the else leg opens up the opportunity for incorporation of its new context into the if-then-else expression. Thus, the developer attaches the following tag to the else leg:

```
(_ON SUBSTITUTIONOFME (_INCOPORATECONTEXT))
```

Now, let us inline the definitions of the **Row**, **Col**, and **W** methods and follow the optimization process.

### 4.3 Composite Folding Optimizations

The specifics of the IA templates are completely defined WITHIN **s** and **sp**. In particular, the size of the

[2] The fully upper case representation of the following expressions is a side-effect of cutting and pasting mechanically generated intermediate data structures into the paper.

point set neighborhood (i.e., the ranges of **?p** and **?q**), must be defined by **s** and **sp**, not by the context in which they are used. Therefore, whenever an IA Template is created by **DSDeclare**, two iterators are automatically created to be used by any code that wishes to iterate over that neighborhood. These iterators will become target program variables. Such iterators are typically represented by gensym symbols such as **piter536** and **quiter538** but for simplicity of exposition, we will call them simply **p** and **q**. When we inline **Row**, **Col**, **W**, **QRange** and **PRange**, these iterators will be bound to the pattern variables **?p** and **?q**, respectively. Inlining these various method definitions causes expression 8 to become ***Expression 9*** (ignoring the **T1** assignment for the time being):

```
(_SUM (P Q)
  (_SUCHTHAT
     (_MEMBER P (_RANGE -1 1))
      (_MEMBER Q (_RANGE -1 1)))
   (A[( I + P) , ( J + Q)] *
    (IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
        THEN 0
         ELSE IF ((P  != 0) && (Q  != 0)) THEN Q
                  ELSE IF ((P  == 0) && (Q  != 0)) THEN (2 * Q )
                           ELSE 0))
```

Since we have just substituted **W**'s body, the transformation that will try to promote the special case test outside of the loop of **P** and **Q** (i.e., **_PROMOTECONDITIONABOVELOOP** ) will be triggered and will immediately run into difficulty. It is within an arithmetic expression which prevents it from fulfilling its enabling conditions. Nevertheless, it does know that there is another transformation (**_INCOPORATECONTEXT**) that (under the appropriate conditions) may be able to incorporate the arithmetic expression into the then and else legs of the if. It triggers this transformation, which transforms expression 9 into ***Expression 10***:

```
(_SUM (P Q)
  (_SUCHTHAT
     (_MEMBER P (_RANGE -1 1))
      (_MEMBER Q (_RANGE -1 1)))
   (IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
       THEN A[( I + P) , ( J + Q)] * 0
       ELSE (A[( I + P) , ( J + Q)] *
                (IF ((P  != 0) && (Q  != 0)) THEN Q
                  ELSE IF ((P  == 0) && (Q  != 0)) THEN (2 * Q )
                           ELSE 0))))
```

Now, **_PROMOTECONDITIONABOVELOOP** can complete its work, further transforming this into ***Expression 11*** (with the **_SUCHTHAT** clause elided to conserve space):

```
(IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
       THEN (_SUM (P Q) ...A[( I + P) , ( J + Q)] * 0)
       ELSE (_SUM (P Q) ...A[( I + P) , ( J + Q)] *
                (IF ((P  != 0) && (Q  != 0)) THEN Q
                  ELSE IF ((P  == 0) && (Q  != 0)) THEN (2 * Q )
                           ELSE 0))))
```

Notice that this transformation has just made two substitutions that will trigger the **_FOLD0** transform on the zero within the then leg and the **_INCOPORATECONTEXT** transform on the weight expression within the else leg. **_FOLD0** creates ***Expression 12***:

```
(IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
       THEN 0
       ELSE (_SUM (P Q) ...A[( I + P) , ( J + Q)] *
                (IF ((P  != 0) && (Q  != 0)) THEN Q
                  ELSE IF ((P  == 0) && (Q  != 0)) THEN (2 * Q )
                           ELSE 0))))
```

and **_INCOPORATECONTEXT** creates ***Expression 13*** (reintroducing the enclosing T1 assignment into the example because we are about to need it):

```
(T1 = (IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
       THEN 0
       ELSE (_SUM (P Q) ...
                (IF ((P  != 0) && (Q  != 0))
                  THEN (A[( I + P) , ( J + Q)] * Q )
                  ELSE IF ((P  == 0) && (Q  != 0))
                           THEN (A[( I + P) , ( J + Q)] *(2 * Q ))
                           ELSE 0)))))
```

At this point, **_CF** finds no more local optimizations that can trigger so it posts the **CFWRAPUP** event which signals the start of cross expression optimizations. The other subexpression **(T2= (BCONV A[I,J] SP))** proceeded through a parallel series of transformations and by the time of the **CFWRAPUP** event, it has the form (***Expression 14***):

```
(T2 = (IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
       THEN 0
       ELSE (_SUM (P Q) ...
                (IF ((P  != 0) && (Q  != 0))
                    THEN (A[( I + P) , ( J + Q)] * P )
                    ELSE IF ((P  != 0) && (Q  == 0 ))
                             THEN (A[( I + P) , ( J + Q)] *(2 * P ))
                             ELSE 0)))))
```

When the **CFWRAPUP** event gets posted, the **_MERGECOMMONCONDITION** transform will fire but it is unable to satisfy its enabling conditions because the two merge candidates are both embedded within assignment statements. Like the **_PROMOTECONDITIONABOVELOOP** transformation, **_MERGECOMMONCONDITION** invokes **_INCOPORATECONTEXT** transformation which succeeds in moving both **T1** and **T2** assignments inside the if-then statement thereby establishing the necessary enabling conditions for **_MERGECOMMONCONDITION** to execute. The sum of these two transforms produces ***Expression15***:

```
(IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
    THEN (T1 = 0; T2 = 0)
     ELSE (T1 = (_SUM (P Q) ...
                (IF ((P  != 0) && (Q  != 0))
```

```
                THEN (A[( I + P) , ( J + Q)] * Q )
                ELSE IF ((P  == 0) && (Q  != 0))
                        THEN (A[( I + P) , ( J + Q)] *
                                (2 * Q ))
                        ELSE 0)))
        (T2  = (_SUM (P Q) ...
                (IF ((P  != 0) && (Q  != 0))
                    THEN (A[( I + P) , ( J + Q)] * P )
                    ELSE IF ((P  != 0) && (Q  == 0 ))
                            THEN (A[( I + P) , ( J + Q)] *
                                    (2 * P ))
                            ELSE 0))))
```

Recall that when the two temporary assignments were generated, the transform recorded the data flow information of **T1** and **T2**. In addition, it tagged each with an expression like

```
(_ON (MigrationIntoBlockOf `T1)  (_SCHEDFOLDFLOWS))
```

In fact, **_MERGECOMMONCONDITION** has just migrated the assignment expressions of **T1** and **T2** into each other's block. At this point, **_SCHEDFOLDFLOWS** wakes up and checks its enabling conditions. Assignments for both **T1** and **T2** must be present at all sources of the data flow to allow relocation of the **B[I, J] = SQRT(T1*T1 + T2*T2)** expression to the source point of each data flow. Further, the right hand side of at least one of the assignments must have simplified to a constant. Otherwise, there is no point to moving the **B[I, J]** assignment because no constant folding would be possible. These enabling conditions are all met in expression 15 and **_SCHEDFOLDFLOWS** produces ***Expression16***:

```
(IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
    THEN (B[I,J] = 0)
    ELSE (T1 = (_SUM (P Q) ...
                (IF ((P  != 0) && (Q  != 0))
                    THEN (A[( I + P) , ( J + Q)] * Q )
                    ELSE IF ((P  == 0) && (Q  != 0))
                            THEN (A[( I + P) , ( J + Q)] *
                                    (2 * Q ))
                            ELSE 0)))
          (T2  = (_SUM (P Q) ...
                  (IF ((P  != 0) && (Q  != 0))
                      THEN (A[( I + P) , ( J + Q)] * P )
                      ELSE IF ((P  != 0) && (Q  == 0 ))
                              THEN (A[( I + P) , ( J + Q)] *
                                      (2 * P ))
                              ELSE 0))
          B[I, J] = SQRT(T1*T1 + T2*T2 )))
```

Now, **_CF** can proceed no further, so it posts the **CFWRAPUPEND** event which will trigger the optimization tags

```
(_ON CFWRAPUPEND  (_UNWRAPIFSMALLCONSTANT P 2)
(_ON CFWRAPUPEND (_UNWRAPIFSMALLCONSTANT Q 2)
```

and this will cause the two loops to be unwrapped, which with partial evaluation produces ***Expression17***:

```
(FOR (I=0; I<M; I++)
  (FOR (J=0; J<N; J++)
    (IF ((I == 0) || (J == 0) || (I == (M - 1)) || (J == (N - 1)))
        THEN (B[I,J] = 0 ; )
        ELSE
          ( T1= ( (A[(I + 1) , (J + 1)] - A[(I + 1) , (J -1)]  +
                  (A[I , (J + 1)]  * 2) - (A[I , (J -1)]  * 2) +
                   A[(I -1) , (J + 1)] - A[(I -1) , (J -1)]) )
            T2 = ( (A[(I + 1) , (J + 1)]  + (A[(I + 1) , J]  * 2) +
                   A[(I + 1) , (J -1)]  - A[(I -1) , (J + 1)] -
                   (A[(I -1) , J] * 2)  - A[(I  -1) , (J -1)] )  )
            B[I,J] = SQRT(T1*T1 +  T2*T2))))))
```

Upon return from **_CF**, the two promotion transformations

```
(_PromoteAboveLoop J  (ConstantExpressionOf  I))
(_PromoteToTopOfLoop J  (ConstantExpressionOf  J))
```

are triggered resulting in expression 2.

### 4.4 Reasoning Over the Optimization Plan

Suppose that one wants to compile expression 1 for a SIMD (Single Instruction/Multiple Data stream) machine, specifically the Pentium II with MMX instructions. AO exhibits a singular advantage in such cases because AO can reason over the abstract optimization plan (as well as the program and domain specific information) to accomplish this purpose. How would this happen for the given example?

The MMX instructions provide the ability to perform vector operations such as parallel add, multiply, and sum of products on two, four, or eight data items at a time. Thus, the optimization plan must reform the target program so that the data (e.g., the image array **a** and the weights of **s** and **sp**) are expressed as data vectors. The way that the AO generator accomplishes this is by re-planing the optimization at the outset so that the optimization phases (and composite folding in particular) will be dealing with components that have a different optimization plan.

In short, the reasoning proceeds as follows. Upon discovering the two **_UNWRAPIFSMALLCONSTANT** tags on the **?p** and **?q** loops, the MMX re-planning transformation recognizes that loop unwrapping would prevent taking advantage of the MMX architecture and so, replaces these tags with **(_ON CFWRAPUPEND (_MMXLOOP))** which will retain the loop structure so that it can be restructured in an MMX friendly way. Next, it recognizes that MMX instructions can only be exploited if **W** of **s** and **W** of **sp** can both be turned into data vectors, ideally at generation time but failing that, at run time. Therefore, it sets about to re-tag **W** of **s** and **W** of **sp** to accomplish this.

To accomplish this, it must 1) keep the body of the **W**'s atomic and not let them be merged into other code, 2) get rid of conditional tests that might interfere with vector operations, 3) create temporary storage for the data vectors of **s** and **sp** and 4) initialize the data vectors, hopefully at generation time rather than execution time. Because of goal 1, the transformation replaces the tag **(_ON SUBSTITUTIONOFME (_INCOPORATECONTEXT))** with the tag **(_ON SUBSTITUTIONOFME (_MAPTOARRAY))** which will prevent the integration of outside context, cause the temporary vector storage to be allocated and generate the initial values for the vectors. Now, it tries to eliminate the special case conditional by splitting the **i**, **j** loop into separate loops for the differ cases, one for each disjunct (e.g., **i==0** && **j** ∈ **[0:(n-1)]**). These loops will be followed by a final loop for the general case (i.e., **( i** ∈ **[1: (m-2)])** && **( j** ∈ **[1: (n-2)])**). Operationally, this generates one loop each to plug zeros into the border elements of the four sides followed by a loop to process the inside elements of the image. With that, the special case test completely disappears. The enabling conditions for this reasoning are 1) recognize that **i** and **j** are loop control variables for some loop in which **W** of **s** (and **W** of **sp**) will be embedded, 2) recognize that **W** of **s** (and **W** of **sp**) are not dependent on the loop control variables that iterate over the pixel neighborhood (i.e., **?p** and **?q**) and 3) recognize that the disjuncts of the special case test (e.g., **(i==0)** or **(i==(M - 1))** ) belong to a category of predicates that modify ranges and thereby can be reasoned over by the specialized loop reasoning rules. Once the enabling conditions are determined acceptable, the transformation replaces the **_PROMOTECONDITIONABOVELOOP** tag with the tag **(_ON CFWRAPUPEND (_SPLITLOOPONCASES)).** When this transformation actually triggers (during **_CF** processing) it will form separate **i**, **j** loops for each disjunct on the then leg and a final general case loop for the else leg. Operationally, for the special case loops, **_SPLITLOOPONCASES** generates separate copies of the **i**, **j** loop and adds a different disjunct into the **_SUCHTHAT** clause. For the general case, it generates a loop copy and adds the negation of the full condition into the **_SUCHTHAT** clause. Then, for each loop, it invokes a set of specialized rules that are designed to recalculate ranges to eliminate the predicates just added. The rules are typified by the following:

**Fixed Index: (∀(?i, ...othervbls...) (?i ∈ [?Low:?High]) (?i==?c) ...otherterms...) : ?body) & (constant ?c) → (∀...othervbls... ...otherterms...: (substitute ?body ((?i ?c)))**

The *fixed index* rule looks for some loop variable (e.g., **?i**) that is asserted to be equal to some **?c** (e.g., **(?i==?c)** ) where **?c** is a constant. If found, the loop is modified to eliminate iteration over **?i** and **?c** is substituted for **?i** in the loop body. This turns a loop like **{∀i,j (i** ∈ **[0:(m-1)]) ( j** ∈ **[0:(n-1)] ) (i==0) : b[i,j] = 0 }** into one like **{∀j ( j** ∈ **[0:(n-1)] ) : b[0,j] = 0 }.** Similar rules are used to clip the ends of ranges or otherwise manipulate the loop ranges. In the case of compiling expression 1 for _MMX, the fixed index rule (above) is used to modify each of the four special case loops and the range clipping rule is used four times to produce the general case loop thus, putting the general case loop into a form in which **_MMXLOOP** will succeed in restructuring the loop's body into an MMX friendly form.

It is worth repeating that this reasoning is performed over the abstraction optimization plan (i.e., the tags) before any optimizations are actually executed. This allows the reasoning logic to reason more globally about the optimization plan tags and their interdependencies. For example, the several separate optimization goals are subtly interdependent – e.g., preventing loop unwrapping, vectorizing **W** of **s** and **sp** (accomplished by **_MAPTOARRAY**), and eliminating case logic within **W** of **s** and **W** of **sp** by splitting the **i**, **j** loop on the cases, and finally, the reformulating the body of the **i**, **j** loop into an MMX friendly form (accomplished by **_MMXLOOP** ). Reasoning is simplified by separating it from optimization execution, treating optimizations in the abstract (i.e., using tag names), reasoning about the global tag interdependencies, and including program and domain knowledge.

## 5. Related research

There is a variety of related research that aims to solve the same or a similar problem or uses similar techniques.

### 5.1 Generation systems

The Draco program generation system shares the domain oriented optimization philosophy of AO, but while it does perform many powerful domain specific optimizations, it does not use AO-like optimization methods or distributed optimization plans for melding and reweaving portions of the code (e.g., loop prefixes). [11, 12] To accomplish similar optimizations, Draco would face a large solution search space or need to resort to programmer intervention and guidance.

Aspect Oriented Programming (AOP) shares with AO the intention and the accomplishment of reweaving code for efficiency. [9] The main difference appears to be that in AOP the optimization plan is a centralized a reweaving algorithm and in AO the plan is factored into a set of separate optimization plan pieces that are distributed over the program in the form of annotation tags. Only when they are brought together through the integration of the reusable program parts does a complete optimization plan exist. This factoring means that the individual optimization pieces are themselves reusable over many past and future reusable code components.

 

Other generation methods are largely orthogonal but complementary to the AO aspect of the generation problem. [2, 14]

### 5.2 Conventional optimization techniques

Each of the optimizations that I have discussed in the AO method have their analogs in the conventional programming language domain [1] with the fundamental difference being that AO achieves the optimizations directly on the domain specific forms without deep program analysis. Conventional optimization, on the other hand, must recover the flow and dependency information via complex analysis techniques that induce large, open-ended searches. Further, conventional optimization often misses opportunities for optimizations because of the difficulty of the associated inference problems (e.g., alias analysis). AO avoids such difficulties by recasting the problem into a form handled by simpler and more effective methods.

The optimization techniques used in APL [7] perform loop fusion like optimization on their built-in data types but provide no way to achieve similar optimization on domain specific types that are not built-in.

### 5.3 Other transformational approaches

Other transformational approaches are often based on some set of general (i.e., not domain specific) technological underpinnings such as functional programming, partial evaluation [8], lazy transformations, or particular transformation algorithms. A good example of this class is the *Deforestation* technique of Wadler [15], which uses a set of transformations to remove intermediate computational forms in a functional programming context, i.e., remove temporary lists and trees.

Such transformation systems emphasize broad generality in the application domain and it is unclear what negative effects such generality will have on the solution search space. Often, the use of just a little domain specific information can vastly simplify the solution search space. But beyond questions of the size of the search space, general optimization techniques are unable to provide the architecture specific variations possible with the domain specific transformations of AO. For example, custom compiling expression 1 for an MMX architecture or a particular display accelerator board would not be possible with broadly general techniques. There is no way to escape the need for domain specific transformations to capture the architecture specific translation techniques required by those target architectures.

## 6. Conclusions

For domain analysis to profitably provide operators and operands that have the ability to both combinatorially amplify programming productivity and provide high performance code, we have to provide optimization strategies that can automatically localize code without engendering the huge search spaces of conventional optimization and transformation methods. Anticipatory Optimization is designed to accomplish this task

## 7. References


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

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

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

[4] Ted J. Biggerstaff, *Anticipatory Optimization in Domain Specific Translation*, International Conference on Software Reuse, June, 1998a.

[5] Ted J. Biggerstaff, *A Perspective of Generative Reuse*, Annals of Software Engineering, 1998b.

[6] Ted J. Biggerstaff, *Composite Folding in Anticipatory Optimization*, Microsoft Research Technical Report, 1998c (forthcoming).

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

[8] Neil D. Jones, *An Introduction to Partial Evaluation*, ACM Computing Surveys, Vol. 28, No. 3, Sept. 1996.

[9] 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.

[10] S. Letovsky, E. Soloway, *Delocalized Plans and Program Comprehension*, **IEEE Software**, May, 1986.

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

[12] 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.

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

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

[15] Philip Wadler, *Deforestation: Transforming Programs to Eliminate Trees*, **Journal of Theoretical Computer Science,** Vol. 73, pp. 231-248, 1990.


 