# RG: A Case-Study for Aspect-Oriented Programming

Anurag Mendhekar, Gregor Kiczales, John Lamping

Technical report SPL97-009 P9710044 Xerox Palo Alto Research Center. February 1997.

# RG: A CASE-STUDY FOR ASPECT-ORIENTED PROGRAMMING

ANURAG MENDHEKAR, GREGOR KICZALES, JOHN LAMPING


XEROX PALO ALTO RESEARCH CENTER*

RG is an image processing system that allows sophisticated image processing operations to be defined by composing primitive image processing filters. An implementation of RG using OOP is easy to do and quite manageable, but its performance is severely lacking. The OOP approach turns out not to be well-suited to addressing the performance problems because the performance issues we want to handle do not respect object or method boundaries and cannot be effectively addressed in a paradigm tied to those boundaries. Aspect-oriented programming is an approach designed to handle complexities arising from such cross-cutting issues. This paper presents a case-study of how this limitation of OOP was overcome using aspect-oriented programming techniques, such that the performance problems were adequately addressed without compromising the original OOP architecture of the system.

## 1. INTRODUCTION

RG (Reverse Graphics) [10] is an image processing system that allows sophisticated image processing operations to be defined by composing primitive image processing filters. The initial implementation of RG was as an OOP library, with images represented by objects and filters invoked by messages. This implementation of RG was quite easy to do and quite manageable in terms of complexity. But this implementation was severely lacking in performance.

While it was easy to trace the cause of these performance problems, and even to anticipate them, the OOP approach turned out not to be well-suited to addressing them. Some performance improvements required changes to be sprinkled throughout the code, resulting in code that tangled the various performance issues along with the basic functionality. Other optimizations couldn't be done at all without breaking the modular structure. Some spanned the client/library boundary, making them impossible to implement in the library at all. In general, the performance issues we wanted to handle did not respect object or method boundaries and could not be effectively addressed in a paradigm tied to those boundaries.

* 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. {anurag,gregor,lamping}@parc.xerox.com

Aspect-oriented programming (AOP) [6, 7, 6] is an approach designed to handle complexities arising from such cross-cutting issues. In AOP, code relating to cross-cutting issues, called aspects, can be written in a way that need not align with the basic component structure. An AOP implementation contains a weaver that combines the aspect code with the code for the basic components to yield executable code that performs as the aspectual code directs.

This paper presents a case study of the application of AOP to the RG image processing system. Section 2 presents the RG system, what the initial OO implementation looked like, and what performance problems it had. Section 3 analyzes the difficulties that arose in trying to address the performance issues using just OOP. Section 4 presents AOP style code to address these problems, and section 5 describes the rest of the AOP implementation, the weaver. Section 6 presents an assessment of the AOP approach in this experiment.

## 2. THE BASE RG SYSTEM

RG is a system developed at Xerox PARC for image processing. It is based on a mode of image processing that has been developed over several years by other PARC Researchers [10]. Programs in RG are written in terms of operations on entire images, rather than working a pixel at a time. All the operations are purely functional; no image is destructively updated. The system is built on a collection of primitive image filters that that take one or more input images to produce an output image. The library defines higher level filters on top of these. Client programmers then define their own filters out of the primitives and the higher level library filters.

The advantage of using this domain model is that it makes it relatively easy to develop complex image processing applications. The system has been used to develop applications such as table recognizers and document genre recognizers with the code being a fraction of the size it would otherwise have been. The resulting applications are easier to maintain and modify, because the core image processing is expressed at a very high level of abstraction.

The primitive filters fall into a few different categories based on the order in which the pixels in the argument images to the filters are processed. Our presentation will focus on two of them:

**Pointwise**: Pointwise operations, such as pointwise addition, combine several images by performing the same operation on each pixel position. The pointwise addition of two images, **AA** and **BB**, is specified by:

```
(pointwise ((a AA) (b BB))  (+ a b))
```
[1]

In this syntax, **a** is a scalar variable bound to a pixel from **AA**, and **b** is a scalar variable bound to the corresponding pixel from **BB**. The last form in the expression indicates what the value of the corresponding pixel in the result should be.

**Translate**: Translate operations are similar to pointwise operations, but shift the pixels in the output image by a specified number of pixels, in a given direction. The following, for example, shifts an image horizontally by two pixels to the right.

```
(translate :x 2 ((a AA))  a)
```

[1] RG is embedded in Common Lisp and the syntax used in this paper reflects this.

```
(defmethod color-by-number-generator ((I image))
   (when! (edges I) I))

   (defmethod edges ((I image))
     (or! (horizontal-edges I) (vertical-edges I)))

   (defmethod horizontal-edges ((I image))
     (or! (right-edges I) (left-edges I)))

   (defmethod left-edges ((I image))
     (not! (equal! I (left-neighbor! I))))

   (defmethod right-edges ((I image))
     (not! (equal! I (right-neighbor! I))))

   (defmethod vertical-edges ((I image))
     (or! (up-edges I) (down-edges I)))

   (defmethod up-edges ((I image))
     (not! (equal! I (up-neighbor! I))))

   (defmethod down-edges ((I image))
     (not! (equal! I (down-neighbor! I))))

   (defmethod left-neighbor! ((I image))
     (translate :x 1 ((a I))
       a))

   ;;; up-neighbor!, down-neighbor!, right-neighbor!
   ;;; are all similar to left-neighbor!

   (defmethod or! ((I1 image) (I2 image))
     (pointwise ((a1 I1)
                 (a2 I2))
       (or a1 a2)))

   ;;; not! and equal! are similar to or!

   (defmethod when! ((Mask image) (Value image))
     (pointwise ((v Value)
                 (m Mask))
```

The other categories include operations to accumulate pixel values along a particular direction, to identify connected regions, to accumulate values within regions, and to return a scalar accumulation of all pixel values.

### 2.1 EXAMPLE

The code in Figure 1 illustrates the use of this system to implement a color-by-number generation application. The task is to take an image consisting of colored regions and return an image that is white except for the outlines

of the regions, with each outline in the color of its region. The input image will consist of color values for each pixel, as will the output image.

This very simple application, **`color-by-number-generator`**, can be expressed in terms of just pointwise and translate primitives. The output image consists of those pixels of the original image that are edge pixels. We define an edge filter, **`edges`** that identifies edge pixels as those pixels that are adjacent to a pixel of a different color. The edge pixel filter is defined in terms of four filters: **`left-edges`**, **`right-edges`**, **`up-edges`**, and **`down-edges`**, that each check for a color difference in one of the four cardinal directions. Each of these compares the pixel color of the original image with that of the image shifted one pixel in the appropriate direction. The pixels that are different are the ones along an edge in that direction.

### 2.2 OBJECT ORIENTED IMPLEMENTATION

A natural implementation of the RG system in an object-oriented language such as Smalltalk, Java™ or CLOS is to represent images as objects and filters as messages. The major part of the storage of an object is an array of pixel values representing the image data. The primitive operations allocate a new image object to hold the result, and then iterate over inputs to fill in the pixel values of the their output, returning the new image object after their iteration is done. New, higher level, operations are defined using the method definition mechanism. As in the figure, the new operations will consists primarily of calls to primitives and other defined operations. The storage of images that are no longer needed is reclaimed by automatic garbage collection.

One of the main reasons for using object-orientation in a system such as RG is that it becomes possible to use alternative image representations without changing the application code. Such alternative implementations might include, for example, an indexed representation where an image is broken up into regions where each pixel has the same value, and then represented by a map from pixels to region identifiers and a map from region identifiers to pixel values. This can offer significant space advantages.

Object orientation works well here in managing the complexity of applications. Successively higher level operations are defined in terms of the lower level operations, without needing to know the details of how they are implemented. The hierarchical composition of the operator functionality naturally aligns with the component structure supported by object-oriented programming.

### 2.3 PERFORMANCE

This naïve implementation of the RG system, however, had three serious (and obvious) performance problems:

1. **Redundancy in computation**: Applications tended to make many redundant calls. Typically this occurs when a higher level filter requests a lower level filter that has already been computed, at the request of a different higher level filter. By not taking advantage of this commonality, the simple implementation did a tremendous amount of redundant computation.

2. **Excess memory turnover**: Since each operation returns a fresh image, images were allocated at a very high rate. This implementation used automatic memory management, and since the images were large, the result was frequent, expensive garbage collection.

3. **Inefficient data cache usage**: Since all the intermediate results were also large images, they quickly filled up the on-chip data cache without ever actually being reused. The result was that accesses to the intermediate results tended to result in too many cache misses and correspondingly slow access times.

```
(defmethod color-by-number-generator ((I image))
   (let ((left-neighbor  (right-neighbor! I))
         (right-neighbor (left-neighbor!  I))
         (up-neighbor    (up-neighbor!    I))
         (down-neighbor  (down-neighbor!  I)))
     (if (already-computed 'color-by-number-generator I)
         (precomputed-result 'color-by-number-generator I)
         (remember-precomputed-result
            'color-by-number-generator
            (let ((result (find-available-storage)))
               (loop-pointwise index
                  ((ipix I)
                   (left-pixel  left-neighbor)
                   (right-pixel right-neighbor)
                   (up-pixel    up-neighbor)
                   (down-pixel  down-neighbor))
                  (if (or (not (equal ipix left-pixel)  ipix))
                          (not (equal ipix right-pixel) ipix)
                          (not (equal ipix up-pixel)    ipix)
                          (not (equal ipix down-pixel)  ipix))
                      (setf (aref result index) ipix)
                      (setf (aref result index) 0))))))
```

*Figure 2 Tangling of the simple RG implementation*

## 3. THE CHALLENGES OF ADDRESSING THE PERFORMANCE PROBLEMS

The strategies to address these performance problems are easy to devise, and straightforward to state.

The first problem, redundant computation, can be addressed with memoization. This is a common technique used in implementing purely functional systems [4]. Each primitive filter can keep a record of the input images it has seen and of what output images result from filtering them. Each invocation of the filter results in a comparison with inputs it has already seen and simply returns the previously computed output if there is a match.

The second and third problems, excess memory turnover and of inefficient data cache usage, are both symptoms of the production of numerous intermediate images. In many cases, intermediate images can be eliminated by fusing loops: applying several primitive filters in a single iteration. Specifically, if one filter produces an image that will be immediately used by another, and if the filters can iterate over the image in the same order, then the computations of the filters can be fused, with each pixel being used by the second filter as soon as it is computed by the first filter. No intermediate array needs to be generated, and there are fewer cache misses. Finally, the overall memory turnover can be further reduced if allocated memory is reused instead of being deallocated and then reallocated.

Since it is easy to state these techniques, we wished for a “smart compiler” to automatically figure out how to apply all of these techniques. It came as no surprise that none was available. While each of these techniques is applied by some compilers in some circumstances, there is no compiler sophisticated enough to apply all of these techniques and, more crucially, to know when to apply each of them. It was up to us to apply these techniques. We found, however, that implementing them within the object oriented paradigm, is not straightforward.

Memoization requires modifying each primitive filter, which is awkward. Much worse, the record of the input and output images from each filter breaks automatic memory management; since every image is now reachable, no images are reclaimed. Custom memory management is not feasible, since the information needed to determine if an image is still useful is scattered throughout the program.

Loop fusion, as a strategy, is not expressible at all in the object-oriented paradigm. And the range of possible fusions is too rich to expect a general purpose automatic compiler to find them. The only way to express these performance improvements in the object oriented paradigm is to abandon the modularity of the original program, to write one large method that incorporates all the optimizations. This results in the kind of tangled code for **`color-by-number-generator`** shown in Figure 2, where all the subsidiary definitions have been unfolded, the pointwise loops have been fused, and memoization code has been added. This kind of code is much harder to read, debug, and modify than the original modular code. And it still breaks automatic memory management. This is, of course, a small example; the optimized code for larger examples is much worse. Constructing them by hand is extremely complex.

## 4. ASPECT-ORIENTED SOLUTION

We have seen that the performance issues crucial to an adequate implementation of the RG system cannot be cleanly addressed in object oriented programming. Procedural programming and functional programming are no better; the performance issues still cross-cut the natural modularization of the basic functionality. What is needed is a programming style that allows the basic functionality to be expressed in accord with its natural modular structure, while letting the handling of performance issues also be cleanly expressed, even though they cross-cut the module structure of the basic functionality.

That is the kind of challenge that Aspect Oriented Programming (AOP) is intended to address. AOP can be used to program systems where the basic functionality can be expressed in a natural component structure, while issues which cross-cut that component structure can be cleanly addressed by other code (cleanly modularized), whose effect will cross-cut the component structure.

Central to this is the concept of "aspect". Kiczales *et al.* [7] compare "aspects" to "components" in the following way:

> With respect to a system and its implementation using a Generalized Procedure based language, a property that must be implemented is:
>
> **A COMPONENT, if it can be cleanly encapsulated in a generalized procedure** (i.e. object, method, procedure, API). By cleanly, we mean well-localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts and GUI widgets.
>
> **An ASPECT, if it can not be cleanly encapsulated in a generalized procedure**. Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects.

### 4.1 WHAT IS NEEDED TO EXPRESS THE SOLUTION

A key step in an AOP solution is to identify the non-component issues and to understand what view of the computation will enable code addressing those issues to be expressed. For RG, the non-component code must be able to express the implementation techniques discussed above: memoization of computations, loop fusion, and memory management. We first look in more detail at what it takes to express each of those techniques.

The implementation of the memoization technique is, roughly, "For every message send invoking a primitive filter, note which filter is being invoked and note the identity of the inputs. If this combination is the same as for some previously seen invocation, use the result of that invocation, rather than recomputing it."

The implementation of the loop fusion technique is, roughly, "For every message send invoking a primitive filter, before computing its arguments, examine each argument and determine whether the loop structure needed to calculate the filter is compatible with the loop structure needed to calculate the argument. In that case, generate a single loop structure that computes both the argument value and the filter value, and replace the original message send with a send to the fused loop."

The memory management technique is, roughly, "Maintain a pool of free arrays. For every message send invoking a primitive filter, allocate the array to hold the output value from the free pool. Once the message send returns, note the identity of each argument array. If that array is not used in any subsequent message send, then place it in the free pool."

Looking at each of these technique descriptions gives a good sense of how the non-component code needs to view the computation. All of them have the phrase "For every message send invoking a primitive filter". These message sends are thus the points in the computation where these techniques want to intervene. In AOP, these are called the join points between the component code and the aspect code.

In addition to seeing the right spots in the computation, each technique needs information about the computation context of those spots to determine the correct behavior. The memoization technique needs to know the identities of the arguments. The loop fusion technique needs to know the loop structure both of the method for the message and for how the arguments were computed, and it needs to be applied before any of the arguments are actually computed so that it can short-circuit those computations. The memory management technique needs to know the identity of the arguments, and it also needs to be able to look into the future well enough to know which arguments will be needed again.

### 4.2 THE LANGUAGES

The AOP approach is to use not only a component programming language, but also to define an additional language or languages that allow the programmer to express the aspectual code that can't fit into the component organization. A weaver then combines the aspect code with the component code to produce an executable.

For our application of AOP to RG, the base language is a subset of CLOS, enriched with constructs for the different categories of primitive RG filters. We adopt one aspect language in which to implement all the performance aspects: a general purpose language (Lisp) whose programs will be given direct access to all the necessary information required to execute the programs. This is appropriate since the performance techniques only need to be described once, for the library. It is thus reasonable to use a relatively general, low level aspect language, rather than the higher level, more specialized aspect languages used in some other aspect-oriented programming applications.

To provide the contextual information that the aspect programs need, they run at compile time and see the join points as a data flow graph of all primitive message invocations in the base program. This is the join point representation. The graph fully unfolds all intermediate message invocations, making the data flows between

primitives readily apparent.[2] Each aspect program is called on each primitive method invocation in the graph and has the opportunity to modify the graph to carry out its performance technique.

The actual structure of the join point representation is a list of *nodes* that each represent a single primitive message invocation. Each node has children that represent the inputs to the invocation. The list itself is sorted in the order in which they will be executed. In particular, the children of every node precede the node itself. To enable the intent of the aspects to be expressed, the data in each node includes information about the call, its inputs, the body of the loop of the callee, *etc*.

Each aspect program must declare when and how it is going to be invoked. For our AOP implementation, there are two kinds of aspect programs. One kind, including the fusion and the memory management aspect programs, has access to the entire list of nodes and can modify it. The other kind, of which the memoization aspect program is an example, has access only to the node that it is being called on. Such an aspect program is necessary if actually creating the whole list might not be practical. For example, if we were to build a call graph of an RG application without doing any sort of node sharing, the size of the list as a whole would become impractically large. In this case, we invoke the aspect program at the point where a node representing a primitive message invocation is about to be created.

## 4.3 THE ASPECT CODE

### 4.3.1 THE MEMOIZATION ASPECT

The behavior of the memoization aspect program is straightforward. It is invoked whenever a call-frame is about to be created. The aspect's behavior is defined by the function **`try-memoization`** below. This function receives two arguments: the actual call expression (the callee and the nodes representing the inputs) and a generator for the node. If the call-expression was seen before, the earlier node is returned; no new node is created. Otherwise, a record is made of the node that corresponds to a certain call-expression.

```
(define-aspect memoization
    :when :on-method-call
    :exec try-memoization)

(defun try-memoization (call-expr call-maker)
  (or (get-memoized-call call-expr)
      (set-memoized-call call-expr (funcall call-maker))))

(defun get-memoized-call (call-expr)
   ;;; if call-expr was seen before, return earlier call
 …)

(defun set-memoized-call (call-expr call)
   ;;; record that call-expr was seen before,and returns call
 …)
```

[2] It is feasible to do this because typical RG base programs have no iterative structure other than what is inside the primitive filters; if the primitive filters are taken to be atomic, an RG program forms a single basic block. If programs did have more than one basic block, the aspect program could be applied separately to each basic block.

### 4.3.2 THE FUSION ASPECT

The fusion aspect is the first to be invoked after the join-point representation is fully available. The function **`try-fusion`** is invoked. This function maps over all the nodes in the list, trying to fuse loops of each node with the loops of its inputs. In the following aspect program, we have illustrated a single fusion possibility: the fusion of pointwise loops with inputs created from pointwise loops. More rules can be added to handle other kinds of fusion.

```
(define-aspect fusion
    :when :first
    :exec try-fusion)

(defun try-fusion (nodes)
  (mapc #'try-fuse-with-inputs nodes))


(defmethod try-fuse-with-inputs ((loop loop-node))
  (let ((new loop))
    (dolist (input (loop-inputs loop))
      (let ((fused  (try-fuse input loop)))
        (if fused (setq new fused))))
    new))

(defmethod try-fuse ((input loop-node) loop)
  (let ((loop-shape  (loop-shape loop))
        (input-shape (loop-shape input)))
   (cond
      ((and (eq (car loop-shape)  'pointwise)
            (eq (car input-shape) 'pointwise))
       (fuse loop input 'pointwise
             :inputs (splice …)
             :loop-vars (splice …)
             :body (subst  …))
      (t nil))))
```

### 4.3.3 THE MEMORY MANAGEMENT ASPECT

The memory management aspect runs after the loop-fusion aspect. While it is not necessary for correctness that this be so, it gives better results. The function that is invoked is **`allocate-memory`**. This function exploits the ordering of the list of nodes in the order of execution. Thus, for a given node, "future" nodes are those that appear after the node in this list and "available" nodes are those that appear before, but are not children of any future node. The allocation for this node can then be made by reusing the space allocated for one of the available nodes.

```
(define-aspect ctma
    :when :second
    :exec allocate-memory)

(defun allocate-memory (nodes)
  (let ((available-nodes nil)
        (future-nodes    nodes))
    (loop while future-nodes do
```

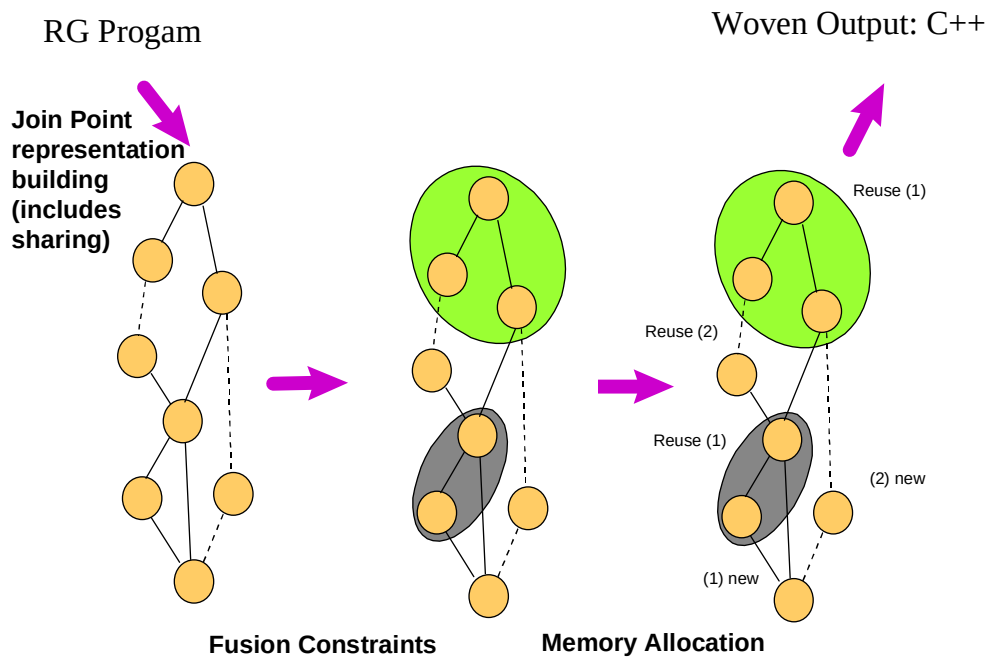


*Figure 3 An overview of weaving in RG*

```
(let* ((node (car future-nodes))
       (previous-allocation
         (there-exists b available-nodes
           (for-all n future-nodes
             (not (immediate-subnode? b n))))))))
 (make-allocation node previous-allocation)
 (setq available-nodes
   (cons node (remove previous-allocation available-nodes)))
 (setq future-nodes
   (cdr future-nodes))))))
```

Looking at the code for the aspects shows why they couldn't be expressed in just the object-oriented approach. Each cross cuts the decomposition of the basic functionality: it acts at each primitive message invocation, not just at particular messages; and it depends on contextual information that wouldn't be available to a normal object oriented method.

### 4.4 THE STRUCTURE OF THE WEAVER

It is the task of the weaver to execute all these aspect description programs against the basic functionality program to produce an output program. The output program will have all the properties with respect to memoization, fusion and memory reuse that we desire.

The design of the weaver is fairly straightforward. There are three big steps in the weaving process. The first step takes the basic functionality program and preprocesses it to uncover join points. It then invokes each aspect

| | Naïve OOP Implementation | Hand Optimized Implementation | AOP Implementation |
|---|---|---|---|
| Timing | > 101 Seconds[3] | 195 milliseconds | 850 milliseconds |
| Number of images allocated. | 158250 | 233 | 43 |
| Source Complexity (In lines of code) | 1500 lines | 35213 lines | 4105 lines (Includes weaver code) |

*Table 1Results of measurements of different implementations of RG*

program based on the `:when` argument in each aspect. Finally, it generates code for the output program. Figure 3 shows an overview of the process.

We build the join-point representation using a simple interpreter that returns message invocation nodes instead of evaluated answers. It invokes the aspect corresponding to `:on-method-call` every time a new node needs to be created. This happens during the evaluation of an application.

When the whole program has thus been interpreted, the return value is a single node that represents the data-flow of the whole program. A walk is made over this graph to determine the sorted list of nodes that forms the join-point representation for the program. This join-point representation is then passed to the next aspect. Since each aspect can destructively modify the join-point representation, the list of nodes is recomputed and re-sorted after every aspect has been invoked.

Finally, the weaver generates output code that corresponds to the woven application. In our case, this code is in C++.

## 5. EXPERIMENTAL RESULTS

This section presents some measurements on a medium-sized image-processing application written using RG. As these numbers show, the AOP architecture for RG is sound and has succeeded in improving the performance of RG without tangling the code. We compare the performance of the AOP implementation with two others. The first is the naïve OOP implementation. The second is the hand-optimized version of the OOP implementation written by researchers on the original RG project [10]. Table 1 presents the results of our measurements.

We use a table recognizer program as our target application. This program identifies what parts of a scanned document are tables and what parts are plain columnar text. The size of the images we work with are $128 \times 128$ pixels.

The performance of the AOP implementation is comparable to the hand-optimized implementation. We attribute the difference in timing between the AOP implementation and the hand-optimized implementation to the fact that the hand-optimized implementation actually performs several other optimizations, notably a packed representation of pixels, that we have not yet added to the AOP implementation. We expect to substantially exceed the performance of the hand tuned implementation after we add those optimizations. Note that the current AOP

[3] This is the time to run the naive implementation on an 8x8 image; the time on a 128x128 image was too slow to measure.

implementation, designed around memory usage as a primary criterion, allocates far fewer images than the hand optimized version. Moreover, the size of the code of the AOP implementation itself is a fraction (about 12%) of the size of the hand-optimized implementation.

## 6. CONCLUSION

We have demonstrated that AOP can help in reducing the code complexity of systems without sacrificing important performance requirements. This case-study has further shown that the weaver for RG has a fairly straightforward structure. It is not hard to imagine a toolkit to support its development without much effort. Such a toolkit would further reduce the complexity of the AOP implementation , bringing it closer to the complexity of the naïve OOP implementation.

## 7. RELATED WORK

AOP has been used to design other systems. Two examples are D [9], a distributed programming framework, and AML [5] a mathematical programming framework. D allows communication strategy and coordination strategy to be encapsulated into aspects. AML's aspects include matrix sparsity and numerical stability.

The transformational style of interpretation of RG's aspects is similar to the idea of transformational programming. In transformational programming, one begins with a simple, provably correct program and then applies correctness preserving transformations manually/semi-automatically and ends up with a much more efficient program that preserves the semantics of the original program [1]. The key difference is that the "aspect" is hidden in the transformation steps, instead of being explicit as it is in the case of AOP. Wadler [12] presents the use of program transformation to reduce intermediate storage in purely functional programs.

The design of our weaver is similar to that of some compile time metaobject protocols [2, 8, 11] which provide a split between a run time base language and a compile-time meta language which can examine the base language program. A key difference, again, is the focus in our project on designing the aspect language with the aspects in mind; exposing the correct joint points and making available the appropriate context.

An implementation of loop fusion of image operations on top of object oriented programming is presented in [3]. That system has a second version of each image operation, which returns a representation of its result consisting of a data structure describing how the image should be computed. Operations on such a representation result in further deferred computations. An explicit request for evaluation causes fused code for the computation to be emitted, compiled, and executed.

## 8. REFERENCES


1. Bird, R.S., *On transformations of programs.* Journal of Computer and Systems Sciences, 1974. **8**: p. 22--35.

2. Chiba, S. *A Metaobject Protocol for C++*. in proc. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 95)*. 1995. Austin: ACM.

3. Cok, D.R. and Cok, R.S. *A chaining and extension mechanism for image processing software*. in proc. *SPIE Image Processing and Interchange: Implementation and Systems*. 1992. San Jose: International Society for Optical Engineering.

4. Hughes, R.J.M. *Lazy Memo Functions*. in proc. *IFIP Conference on Functional Programming Languages and Computer Architecture*. 1985. Nancy: Springer Verlag.

5. Irwin, J., Loingtier, J.-M., Gilbert, J., Kiczales, G., Lamping, J., Mendhekar, A., and Shpeisman, T., *Aspect-Oriented Programming of Sparse Matrix Code*, *Submitted for Possible Publication, OOPSLA 97*. 1997.

6. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C., Maeda, C., and Mendhekar, A. *Aspect-Oriented Programming*. in proc. *POPL Workshop on Domain Specific Languages.* 1997. Paris.

7. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C., Maeda, C., and Mendhekar, A., *Aspect-Oriented Programming, Submitted for possible publication to OOPSLA 97*. 1997.

8. Lamping, J., Kiczales, G., Rodriguez Jr., L.H., and Ruf, E. *An Architecture for an Open Compiler*. in proc. *IMSA'92 Workshop on Reflection and Meta-level Architectures*. 1992.

9. Lopes, C.V. and Kiczales, G., *D: A Language Framework for Distributed Programming*, *Submitted for Possible Publication, OOPSLA 97*. 1997.

10. Mahoney, J.V., *Functional Visual Routines*, TR SPL95-069, 1995, Xerox Palo Alto Research Center: Palo Alto.

11. Mendhekar, A., Kiczales, G., and Lamping, J., *Compilation Strategies as Objects*, in *Informal Proceedings on the OOPSLA 1994 Workshop on Object-Oriented Compilation*. 1994.

12. Wadler, P., *Deforestation: transforming programs to eliminate trees*. Theoretical Computer Science, 1990. **73**: p. 231--248.