Generation Flexibility versus Performance

Ted J. Biggerstaff

Microsoft Research

One Redmond Way, Redmond, WA. 98052-6399

Tel: (425) 936-5867, fax: (425) 936-7329

Email: tedb@microsoft.com

URL: http: //research.microsoft.com/~tedb/index.html

Abstract

Defining domain specific abstractions for generator systems leads to a quandary between choosing abstractions that exhibit powerful programming amplification through the combinatorial opportunities provided by composition, and choosing abstractions that can be easily transformed into high performance code. Most generation systems opt for high levels of abstraction to achieve programming amplification and as an added benefit get safety, understandability, and several other -ilities. As a consequence, the performance of their generated code is often compromised. My hypothesis is that a new generator architecture is needed to achieve both high levels of abstraction and high performance code. A generator based on such an architecture has been implemented in Common LISP. It is called the *Anticipatory Optimization* based generator because it allows the component and transform writers to **anticipate** the kinds of optimization opportunities that might arise and to prepare an abstract, distributed plan that attempts to achieve them. Based on a small number of examples that we have tested, the approach appears promising, allowing high levels of abstraction, flexibility, and performance. (See Biggerstaff98a-b.)

Keywords: Domain specific, program generation, transformations.

Workshop Goals: List any goal you have, such as to solve a particular problem.

Working Groups: Groups on subjects like generators and transformation systems.

1 Background

Program generators compile high level, compact, (and usually domain specific) languages into conventional programming languages like C or Java thereby improving programming productivity, code safety, ease of understanding, and so forth. The same properties that make such domain specific language (DSL) representations appealing make compiling them into high performance code difficult. The performance problems arise because DSLs tend to delocalize performance related elements of the target code and introduce high levels of redundancy into the target code. The various approaches to optimizing the generated code (e.g., conventional optimization, optimizing transformations, specialization, etc.) all have practical problems (e.g., huge search spaces) that call into question their feasibility.

2 Position

The hypothesis of this paper is that a new architecture for generators is required to overcome this problem in a practical way. The essential feature of this new architecture is the ability to attach tags to any element of the Abstract Syntax Tree (AST) that represents a DSL expression and use those tags to optimize the generated code. The tags can be thought of as mostly a distributed optimization plan, but they also include translation status information (e.g., inferred types). The generator both manipulates the tags (i.e., reasons about the tags, the domain information, and the program constructs) and eventually executes them by calling the optimizing transformations associated with them (e.g., the PROMOTECONDITIONABOVELOOP transformation, which tries to move a condition test outside of a loop). Through this tag driven optimization process delocalized code gets relocalized and redundant code gets shared. Some tags depend on optimization events (e.g., substitution of a tagged expression) for their triggering condition, which may, in turn, trigger a whole cascade of other opportunistic transformations.

3 Approach

3.1 Example

An example of a domain specific programming expression is illustrated by expression for Sobel edge detection in bitmapped images.

Dsdeclare image a, b form (array m n) of bwpixel;

b = [(a ⊕ s)^2 + (a ⊕ sp)^2]$^{1/2}$;

where **a** and **b** are **(m X n)** grayscale images and ⊕ is a convolution operator that applies the 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 OO classes that define the specifics of the pixel neighborhoods (i.e., the neighborhood sizes and shapes, the weights to be associated with each position in the neighborhood, and any special processing such as the special case when **s** or **sp** are hanging off the edge of the image). The convolution operator iterates over the image **a** performing a sum or products of **a[i,j]** and all of its neighboring pixels. The neighborhoods are defined by **s** and **sp**. For this example, they are defined as

`s [(-1:1), (-1:1)] = {{-1, 0, 1}, {-2, 0 , 2}, {-1, 0, 1}}` and

`sp [(-1:1), (-1:1)] = {{-1, -2, -1}, {0, 0, 0}, {-1, -2, -1}}.`

For a single CPU Pentium machine without MMX instructions (which are SIMD instructions that perform some arithmetic in parallel), the AO generator will produce code that looks like

```
for (i=0; i < m; i++)    /* Version 1 */
        {im1=i-1; ip1= i+1;
                for (j=0; j < n; j++)
                        { if(i==0 || j==0 || i==m-1 || j==n-1) /*Off edge*/
                        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 requires 62 large grain transformations and is produced in a few tens of seconds on a 400 MHz Pentium. I believe that most of this time is due to the experimental nature of the implementation. By redesigning the implementation data structures, some search algorithms that are exponential in the height of the expression tree will become constant time operations (i.e., a single pointer dereference) and I expect that this will drop the generation times near the range of optimizing compiler times. In any case, the AO generator can write the code a good deal faster than I can.

In contrast, if the machine architecture is specified to be MMX, the resultant code is entirely different from version 1:

```
{int s[(-1:1), (-1:1)]={{-1, 0, 1}, {-2, 0 , 2}, {-1, 0, 1}};/* Version 2 */
int sp [(-1:1), (-1:1)]={{-1, -2, -1}, {0, 0, 0}, {-1, -2, -1}};
for (j=0; j < n; j++) b[0,j] = 0 ;
for (i=0; i < m; i++) b[i,0] = 0 ;
for (j=0; j < n; j++) b[(m-1),j] = 0 ;
for (i=0; i < m; i++) b[i,(n-1)] = 0 ;
{ for (i=0; i < m; i++)
        { for (j=0; j < n; j++)
                {t1 = UNPACKADD(PADD2 (PADD2 (PMADD3 (&(a[i-1, j-1]), &(s[-1,
-1])) ,
                                                PMADD3 (&(a[i, j-1]), &(s[ 0,
-1]))),
                                        PMADD3 (&(a[i+1,j-1]), &(s[ 1, -1])));
                t2 = UNPACKADD(PADD2 (PMADD3 (&(a[i-1, j-1]), &(sp [-1, -1]))
, PMADD3 (&(a[i+1, j-1]), &(sp [ 0, -1]))) ) );
                b[i,j] = sqrt(t1*t1 + t2*t2);}}}
```

Where the functions UNPACKADD, PADD$_2$, and PMADD$_3$ correspond to MMX instructions and are defined as PMADD$_3$ ((a0, a1, a2) , (c0, c1, c2)) = (a0*c0+a1*c1, a2*c2 +0*0), PADD$_2$ ((x0, x1) , (x2, x3)) = (x0+x2, x1+x3), PMADD$_3$ ((a0, a1, a2) , (c0, c1, c2)) = (a0*c0+a1*c1, a2*c2 +0*0), and UNPACKADD((x0, x1)) = (x0+x1). All lend themselves to direct translation into MMX instruction sequences. **`s`** and **`sp`** have become pure data arrays to optimize the use of the MMX instructions. Notice, that the special case test (i.e., is the template hanging over the edge of the image?) has completely disappeared. Transformations that perform simple inferences have split the main loop on that test turning the single loop in the previous version into five loops. Four loops plug zeros into the four edges of the image (i.e., the new form of the special case processing) and one loop processes the inside of the image (i.e., the non-special case processing). The fundamental difference in the derivation of the two versions is in the tag driven optimization phase. Up to that stage, the transformations that fire are exactly the same. The form of the target program in the two cases just before tag driven optimization is exactly the same except for the tags.

3.2 Operation

How does the AO generator accomplish this? The generator is a multi-phase, transformation system. The early phases use *pattern directed transformations* (i.e., transformations that trigger based on code patterns) to translate the high level operands into lower level conventional programming constructs. For example, these transformations will refine images into pixels, pixels into channels, and channels into integers. In the course of this, they also create, place, and fuse the implied looping constructs. These transformations may also perform opportunistic optimizations. For example, in the convolution example above, a pattern directed transformation recognized that the expressions $\mathbf{(a \oplus s)^2}$ and $\mathbf{(a \oplus sp)^2}$ allow a reduction in strength optimization to **(t1 * t1)** and **(t2 * t2)** , if the temporary assignments **t1 =(a ⊕ s)** and **t2 =(a ⊕ sp)** are created and moved out of line.

The later phases use *tag directed transformations* (i.e., transformations that trigger based on tags attached to the program) to incorporate operator definitions (e.g., the convolution operator and the methods of **s** and **sp**), reorganize the resulting forms, and simplify the resulting code. These reorganizations may cause optimization events (e.g., substitution of a subtree) that further trigger event-based transformations, i.e., tags with explicit triggering conditions. These may cascade to completely reorganize the program. For example, substitution of the convolution operator definitions in the earlier example starts a cascade of transformations. It moves the neighborhood loop into the then and else legs of an if-then-else expression that computes the weights, moves the multiplication of the **a[i,j]** pixel into the then and else legs (recursively), and triggers constant folding that reduces the loop in the then leg to zero.

These tags may be pre-positioned on reusable library components (e.g., on the definition of the convolution operator) in anticipation of potential optimizations. They are also added and deleted by other transformations in the course of generation.

4 Comparison

This work bears the strongest relation to Neighbors work. The main differences are 1) the fact that the AO pattern directed transformations are organized into an inheritance hierarchy which guides the choice of which transformations to try, and 2) the use of the tag directed approach to program optimization. Neighbors uses pattern directed transformations during his optimization phases.

The work bears a strong relationship to Kiczales' Aspect Oriented programming at least in terms of its objectives. The optimization machinery appears to be quite different in the two approaches. Kiczales' optimization mechanism seems to be centralized and the optimization algorithm itself does not appear to be manipulated by the transformations. In contrast, the AO generator's tags are distributed over the program and they undergo many transformations as the generator reasons about the domain, the program, and the optimization tags. The tags come and go during the execution of both types of transformations although the pattern directed transformations manipulate them more frequently and purposefully than the tag directed transformations do.

The work is largely orthogonal and complementary to the work of Batory. However, both make strong use of domain specific components and information in the course of their operation.

The AO generator and Doug Smith's work are similar in that they make heavy use of domain specific information in the course of generating code. They differ in the machinery used. Smith's relies much more heavily on inference machinery than does AO. The reasoning that AO does is narrowly purposeful and is a somewhat rare event (e.g., the transformation that splits the loop in the MMX example above does highly specialized reasoning about the loop limits). However, partial evaluation (a form of inference) is used quite heavily in the AO generator, which is how three level if-then-else statements get reduced to expressions like "**`a[im1, j] * (-2)`**".

References

[Batory93] Batory, Don, Singhal, Vivek, Sirkin, Marty, and Thomas, Jeff, "Scalable Software Libraries," *Symposium on the Foundations of Software Engineering*. Los Angeles, CA, December, 1993.

[Biggerstaff98a] Biggerstaff, Ted J., "Anticipatory Optimization in Domain Specific Translation," *International Conference on Software Reuse*, Victoria, B. C., Canada, June 1998, pp. 124-133.

[Biggerstaff98b] Biggerstaff, Ted J., "Composite Folding in Anticipatory Optimization," *Microsoft Research Technical Report MSR-TR-98-22*, June 1998, pp. 10.

[Kiczales97] Kiczales, Gregor, Lamping, John, Mendhekar, Anurag, Maede, Chris, Lopes, Cristina, Loingtier, Jean-Marc and Irwin, John "Aspect Oriented Programming," *Tech. Report SPL97-08 P9710042*, Xerox PARC, 1997.

[Neighbors89] Neighbors, James M., "Draco: A Method for Engineering Reusable Software Systems." In Ted J. Biggerstaff and Alan Perlis (Eds.), *Software Reusability*, Addison-Wesley/ACM Press, 1989, pp. 295-319.

[Smith91] Smith, Douglas R., "KIDS-A Knowledge-Based Software Development System," in *Automating Software Design*, M. Lowry & R. McCartney, Eds., AAAI/MIT Press, 1991, pp.483-514.

Biography

Ted Biggerstaff (tedb@microsoft.com) One Redmond Way, Redmond, WA. 98052, http: //research.microsoft.com/~tedb/index.html /

Dr. Biggerstaff got his Ph.D. from University of Washington and has been largely without adult supervision since then. He has done research on program generation in the early years and later on development tools. He was a Director at MCC where he led a group doing reuse, program understanding, and hypertext research. At Microsoft his roles have been at various times program manager, research ambassador, and most recently, a researcher on program generation. He has spent a good deal of time lately playing his favorite video game -- The Common LISP development system.

He has written two books and many papers. He is a frequent keynote speaker, invited speaker, or conference panelist.