A Class Specific Optimizing Compiler

Michael D. Sharp
Steve W. Otto

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-010

April 1993

A Class Specific Optimizing Compiler

Michael D. Sharp and Steve W. Otto
Oregon Graduate Institute of Science & Technology
sharp@cse.ogi.edu, otto@cse.ogi.edu

March 21, 1993

Abstract

Class specific optimizations are compiler optimizations specified by the class implementor to the compiler. They allow the compiler to take advantage of the semantics of the particular class so as to produce better code. Optimizations of interest include the strength reduction of class::array address calculations, elimination of large temporaries, and the placement of asynchronous send/recv calls so as to achieve computation/communication overlap. We will outline our progress towards the implementation of a C++ compiler capable of incorporating class specific optimizations.

1 Introduction

During the implementation of complex systems in C++, particularly numerical ones, the implementor typically encounters performance problems of varying difficulty. These difficulties usually relate to the lack of semantic understanding the C++ compiler has of the user defined classes. This problem was recently studied in [2] where the potential solution of class based optimizations was put forth.

A class based optimization is an optimization which makes use of semantic information normally not known to the compiler. These optimization rules are specified by the user as part of the class description and they are dynamically linked to the compiler's standard optimizer. While the notion of a rule directed optimizer is not new [3] they are not widespread. The authors believe this is the first time the optimization rules have been user specified for the C++ language.

After introducing two example optimizations, this paper will focus on some of the issues relating to the construction of a system implementing class based optimizations. The issues discussed relate mainly to optimization specification, detection of applicability, and application.

```
class Matrix {
        public:
                Matrix();
                Matrix& operator=(const Matrix&);
                Matrix& operator+=(const Matrix&);

                friend Matrix& operator+(const Matrix&,const Matrix&);
        };

        main(){
        Matrix A, B, C, D;
                // ...
        A = B + C + D; // Fragment 1
                // ...
        A = B; A+ = C; A+ = D;  // Fragment 2
                // ...
        }
```

Figure 1: Matrix code fragments

2 Sample optimizations

Throughout this paper two optimizations will be used as examples. The first is the temporary variable elimination optimization, and the second is strength reduction combined with induction variable analysis in a general array iterator. The later construct is an extension the authors have made to the C++ language.

2.1 Temporary variable elimination

In numerical computations it is often advantageous to optimize a program for the amount of memory used. One of the easiest ways to optimize a program for minimal memory usage is to eliminate large temporary variables. We will use the example of matrix calculations to demonstrate the point. The two code fragments appearing in Figure 1 show the value of this optimization. The first fragment requires a temporary matrix while the second avoids this by performing the calculation in place. Ideally the transform from the first to second fragment would be handled by a matrix class specific optimization.

The above optimization applies only if the '+' operator is at the root of the expression tree. The same type of optimization will also apply for other overloaded operators such at $*$,/, and $-$. This is not true in general, but if a user overloads $\oplus$ then the operation is usually one that has similar characteristics

$X.iterate\ i$ over $[0 : 100 : 1]$\$ {
$X.elem(i) = a * X.elem(i) + y.elem(i);$
}

Figure 2: One dimensional array iterator

to the integer $\oplus$ operator.

We now consider what happens in the optimization of a general expression containing several operators. The optimization rule is continually applied to the expression tree starting at the root. If the operator at the root of this tree is TVE (the ability to express the expression without temporary usage at this level) the single statement is split into two statements (the = and the + =) as was done in Fragment 2 of Figure 1. The optimization is then applied to each statement in turn. The statements continue to split into more statements as long as the root operator has the TVE property. If the root operator for a statement is not TVE then a temporary (of potentially large size) must be created.

2.2 Optimizing abstract array iterators

Consider a partitioned array container class as described in [6]. The partition types that are supported are block and block cyclic.

In FORTRAN, access at array elements inside of **do** loops is very efficient. This is possible since FORTRAN does not have the pointer aliasing problems of C and C++, and the semantics of the **do** loop is simpler than those of **for**. As a result FORTRAN compilers are able to perform induction variable analysis and strength reduction so that array address calculations are done efficiently. While there are C++ compilers, g++ for example, capable of such optimizations, this is not the norm. One of our goals is to provide such a strength reduction optimization on a class by class basis. Using this approach it is possible to avoid illegal applications and to guarantee the optimization will be applied without relying on the underlying compiler to implement it.

Consider the simple example of an iterator for a one dimensional array in figure 2. If X is a block partitioned array this iterator might be implemented along the lines of figure 3, and if X is block-cyclic partitioned, the iterator might be implemented as in figure 4.

Clearly the situation becomes complex for multi-dimensional, block-cyclic partitioned arrays. With proper optimizations for array iterators, the coding complexity of multi-dimensional computations can be reduced. General iterators also expose opportunities for additional optimization due to the less restrictive nature of the control structure. That is, since a precise ordering of the iteration

```
for (i = X.start(0); i < X.end(0); i++) {
    *(X.base + i) = ...
    }
```

Figure 3: 1-D block partitioned array iterator

```
for (l = 0; l < X.numBlocks(0); ++l) {
    for (i = X.start(0, l); i < X.end(0, l); ++i) {
        *(X.base[l] + i) = ...
        }
    }
```

Figure 4: 1-D block-cyclic partitioned array iterator

space is not specified by the programmer the optimizer has more flexibility in loop restructuring.

3 Optimization specification

Two of the most difficult technical problems in the implementation of class based optimizations are defining a language in which to describe general optimizations, and the implementation of the pattern matching routine which detects when to apply optimizations. What is presented in this and in the next section are not complete answers to these difficult problems, rather the current direction of research of the authors.

In attempting to define a language to describe general optimizations there are a number of issues to be considered. It must be possible to not only describe the syntactic pattern to match, but to also specify the semantics, and dependencies of this code. Any optimization triggering heuristics must also be specifiable in this language.

The syntactic patterns to be matched may not necessarily be contiguous. It is quite reasonable to expect user defined optimizations to require the ability to skip past statements searching for some matching condition, or to require a certain set of conditions for an arbitrarily long list of commands. For example, the iteration optimizations discussed earlier require the examination of the entire loop body.

A language for the specification of optimizations called GOSPEL is presented in [7]. This language expresses optimizations in terms of both the general pro-

gram structure to be matched as well as the data dependencies necessary for the optimization to result in semantically correct code. Currently we are implementing optimizations at a much lower mechanical level (figure 5). Future research includes the definition of a language similar to GOSPEL, but more closely tied to C++.

An ideal form of specification would be C++ extended with inspiration from programming logics. In such a language the general syntactic form of the optimization could be specified by fragments of C++, while the data dependence and any heuristics could appear in embedded assertions. It would surely be the most useful representation since optimizations would then be specified more by partial code examples and a few language extensions then by another language altogether. Figure 6 shows a possible form of a loop interchange optimization.

The current design of our optimizer is quite similar to what one would use to implement an optimization in a traditional compiler. It is very dependent on the internal representation of the code and the writer of such an optimization must have knowledge of this representation. While neither of the authors is satisfied with this as a final goal, it is felt to be a good intermediate step to prove the concept of class based optimizations.

4 Optimizer implementation

The optimizer's implementation is greatly complicated by the fact that before an optimization can be applied, the associated pattern of un-optimized code must be located in the internal representation of the program. In the past various code generated and peephole optimizers [1, 4, 5] have done this, but either always on small contiguous patterns, or, if an attributed grammar is used, with restrictions on the use of attributes. In the case of this optimizer it must be possible to match noncontiguous patterns, as they were discussed in the previous section, and to add additional attributes (derived from operations on the compiler supplied ones) based on the needs of the optimization.

Another complicating factor for the implementation of the optimizer is the determination of the optimization ordering. Usually this is determined by the compiler architect, however since the actual optimizations are now being supplied by the class designers, it is quite conceivable that the ordering of optimizations will play a role in the efficiency of the optimized code. Ordering problems will hopefully be minimal since optimizations are triggered by class occurrences, but an ordering mechanism should still be explored. Certainly such a mechanism will depend heavily upon the user's application and should be specified by the user if the default ordering is not acceptable.

```
AssignmentPtr=Find(Assignment);
while (AssignmentPtr) {
   // check the type of this assignment
   if(Type(AssignmentPtr)==MatrixClass) {
    // check for the pattern B+C on the right
    // hand side of the assignment where B and C
    // are any subexpression.
    ExprPtr=RightHandSide(AssignmentPtr);
    if(Operator(ExprPtr)==OP_Plus) {
     // break A=B+C into A=B;A+=C
     // since a match was found this statement should be
     // re-processed in hopes of finding another.
     // AssignmentPtr should not be changed before the
     // next loop iteration.
     Tree construction code omitted for brevity. The
     newly constructed tree is pointed to by APlusCTree
     RightHandSide(AssignmentPtr)=LeftOperand(ExprPtr);
     InsertStatementAfter(AssignmentPtr,APlusCTree)
     }
    else {
     // didn't find a pattern match.
     // move to the next assignment statement
     AssignmentPtr=FindNext(Assignment,AssignmentPtr);
     }
    }
   else {
    // didn't find a pattern match because
    // the class type was wrong
    AssignmentPtr=FindNext(Assignment,AssignmentPtr);
    }
   }
```

Figure 5: Current specification form for temporary elimination

```
$pattern

    for ($1,$2,$3) {
        for ($4,$5,$6) {
            $A;
            }
        }

    ${
        /* check for dependence between invariants */
        /* ($1->Statement is the statement containing */
        /* the fragment represented by $1) */
        if Dependence($1->Statement,$4->Statement,any) fail;

        /* check for (<,>) dependence between */
        /* two statements in the loop body */
        forAllStmt($A,$7) { /* for all individual statements $7 in $A */
            forAllStmt($A,$8) {
                /* check dependence for legality of optimization */
                if (Dependence($7,$8,"(<,>)")) fail;
                }
            }
        $}

$optimization
    for ($4,$5,$6) {
        for ($1,$2,$3) {
            $A;
            }
        }
```

where:

$A is a meta variable representing zero or more statements

$1-$n are meta variables representing components of a statement

Figure 6: Loop interchange specification

5 Current Status

At the time of writing, the authors have a working C++ to C++ optimizer which was custom built for this project. This is felt to be of great worth due to the avoided additional complexity of layering such an optimizer on top of a public domain compiler which wasn't designed with such capabilities in mind.

The optimizer was implemented using a tool, developed by one of the authors, to describe complex attribute relationships and structures. This tool allowed a relatively quick implementation of a very memory efficient internal representation of C++. Since all of the attributes of this representation are managed and mapped by this tool, there is great flexibility in our optimizer when it comes to adding to the internal program representation.

Work is currently under way to define the optimization specification language, as well as implement the pattern matcher. As was stated earlier, the current approach is a very mechanical and strongly dependent upon the internal representation of the program being optimized. It is the authors' goal to evolve this into a much higher form in the hopes of hiding many of the details of the compiler implementation.

References

[1] A. V. Aho, M. Ganapathi, S. W. K. Tjiang, Code Generation Using Tree Matching and Dynamic Programming *ACM Transactions on Programming Languages and Systems*, 11(4):491-516 (1989)

[2] I. G. Angus, Applications Demand Class-Specific Optimizations: The C++ Compiler Can Do More, 1993 Object-Oriented Numerics Conference, April 25-27, 1993, Sunriver, Oregon

[3] J. W. Davidson, D. B. Whalley, Quick Compilers Using Peephole Optimization, *Software–Practice and Experience*, 19(1):79-97 (1989)

[4] M. Ganapathi, C. N. Fischer, Affix Grammar Driven Code Generation, *ACM Transactions on Programming Languages and Systems*, 7(4):560-599 (1985)

[5] R. S. Glanville, S. L. Graham, A new method for compiler code generation, *Proceedings of the fifth annual ACM Symposium on Principles of Programming Languages*

[6] S. W. Otto, Parallel array classes in lightweight sharing mechanisms, 1993 Object-Oriented Numerics Conference, April 25-27, 1993, Sunriver, Oregon

[7] D. Whitfield, M. L. Soffa, Automatic Generation of Global Optimizers, *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*