Automatic Generation of Global Optimizers*

Deborah Whitfield and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

This research has developed an optimizer generator that automatically produces optimizers from specifications. Code optimizations are expressed using a specification language designed for both traditional and parallelizing optimizations, which require global dependence conditions. Numerous optimizers have been produced from a prototype implementation of the generator. The quality of code produced using the generated optimizers compares favorably with that produced by hand coded optimizers. The generator can be used as a phase in a compiler or as an experimental tool to determine the effects of various optimizations and to tailor optimizations. Experiments indicate that optimizations interact in practice and that different orderings of optimizations are needed for different code segments of the same program. Experiments found that the cost-benefit ratio of some optimizations is quite large and in some cases can be reduced by careful specifications of the optimizations or different implementations.

1. Introduction

Although traditional global optimizations have long been applied to program code, the cost of implementing these optimizers remains high. We have reduced the cost of implementing optimizers by providing a General Optimization Specification Language, (GOSpeL) and an optimizer generator (GENesis) that allows users to generate a wide variety of global optimizers from compact, declarative specifications. The optimizers thus produced are useful in conventional compilers but are particularly useful when experimenting with optimizations and when compiling for parallel machines, where it maybe unclear which transformations to use and how to order them.

To use GENesis, the optimizations under consideration for application on program code are expressed in GOSpeL by the user. For each optimization, GOSpeL requires the specification of the preconditions and the actions to optimize the code. In addition, the interactive capability that the user desires with the optimizer can be specified. The preconditions consist of the code patterns and the global dependence information needed for the optimization. The code patterns express the format of the code, and the global information describes the control and data dependences that are required for the specified optimization. The actions take the form of primitive operations that occur when applying code transformations. The primitives are combined to express the total effect of applying a specific optimization. For generality, GENesis assumes a high level intermediate representation that retains the loop structures from the source program. The user can specify whether the optimization should be applied automatically (e.g., traditional optimizations) or should be applied at the user's direction (e.g., parallelizing transformations). The higher level intermediate code allows the user to interact at the source level for loop transformations, typically applied for parallel systems.

With an optimizer generated by GENesis, the user can experimentally investigate the performance of the optimizations on program code for the system under consideration. The cost and expected benefit of various optimizations can be compared on production code. Optimizations that are not effective can be removed and other optimizations can be added by simply changing the specifications and rerunning GENesis, producing a new optimizer. The decision as to the order in which these optimizations should be applied can be easily investigated. New optimizations can be created or existing optimizations tailored to the system and easily incorporated into an optimizer.

* This work was partially supported by the National Science Foundation under Grant CCR-8801104 to the University of Pittsburgh.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

Currently, we have used GOSpeL to specify approximately twenty optimizations found in the literature and have been successful in specifying all optimizations attempted. In order to test the viability and robustness of this approach, we implemented a prototype for GENesis and have produced a number of optimizers. Using these optimizers, the impact of ordering optimizations as well as the cost and benefit of the optimizations have been investigated experimentally. In the next section, examples of the specifications of optimizations in GOSpeL are given as well as an overview of the language. Section 3 describes the design and implementation of GENesis. Experimental results are presented in Section 4. The paper concludes with a section on related work.

2. Description of GOSpeL

GOSpeL permits the uniform specification of both traditional, sequential optimizations and parallelizing transformations by using common constructs. Control, anti, flow and output data dependences are used in this work, as these dependences are needed to specify parallelizing optimizations and can also be used to express traditional data flow for sequential optimizations. Thus the use of a common data dependence notation in the specification is a step toward unification of parallelizing and traditional optimizations.

A flow dependence (S_i δ S_j) exists between a statement S_i that defines a variable and each statement S_j that uses the definition from S_i. An anti-dependence (S_i $\bar{\delta}$ S_j) occurs between statement S_i that uses a variable that is then defined in statement S_j. An output dependence (S_i δ^o S_j) exists between a statement S_i that defines or writes a variable that is later defined or written by S_j. A control dependence (S_i δ^c S_j) exists between a control statement S_i and all of the statements S_j under its control. In other words, if S_i is an **IF** condition then all of the statements within the **THEN** and the **ELSE** are control dependent on S_i. The concept of data direction vectors for both forward and backward loop-carried dependences of array elements is also used in this work. When determining the data dependences within loops, it is necessary to examine the direction of the dependence[10]. Each element of the data dependence vector consists of either a forward, backward, or equivalent direction represented with <, >, or =, respectively. An * is used when any of the three directions can apply (omitting the direction vector produces the same result). The number of elements in the direction vector corresponds to the loop nesting level of the statements involved in the dependence.

We first present examples of specifications and then highlight the features of the language by considering the syntax and semantics of the components of an optimization specification. A BNF grammar was developed for GOSpeL, and a subset appears in the Appendix of this paper. The grammar is used to construct well-formed specifications and also to implement an optimizer through GENesis.

The specification of an optimization has three sections: type, precondition and action sections, where the precondition is sub-divided into two parts. The type section specifies the required code element types. The precondition section includes both the code format specification and the dependences that are needed. The data dependence specification involves the description of statement and operand dependences, direction vectors, and any necessary membership qualification. The action section specifies the primitive actions that perform the code transformations. The general independence of the specification from the underlying implementation allows for the implementation of GOSpeL at any level, including source level. However, the general form that a statement may take is needed to delimit statement components.

The assumed representation of an assignment statement for this implementation of GOSpeL is:

opr_1 := opr_2 opc opr_3

The number of operands could be modified to reflect other program representations. Figure 1 demonstrates a GOSpeL specification of Constant Propagation (CTP).

The Code_Pattern keyword in the PRECOND section of Figure 1 specifies an occurrence of a statement that assigns a constant, represented by the second operand. The data dependence conditions given in the Depend section of PRECOND expresses the conditions that must exist before applying the optimization. For CTP, the condition is to locate a use of the defining variable of Si (i.e., opr_1) at statement Sj (in operand position **pos**) and ensure that there are no other definitions that reach the use. If such a statement is found then the actions expressed in the action section are performed. The action is to modify the use at Sj to be the constant found as the second operand of Si. **Modify** is just one of the primitive actions allowed in GOSpeL.

Next consider the specification of the parallelizing optimization Loop Interchanging (INX) found in Figure 2. In the Code_Pattern section, **any** specifies an occurrence of two tightly nested loops L1 and L2. Two loops are tightly nested if one surrounds the other without any statements between them[14]. The data dependence condition in the Depend section expresses two conditions. First, it ensures that the loop headers are invariant with respect to each other by checking for a flow dependence. Also, the Depend section expresses that there are no pairs of statements in the loop with a flow dependence and a (<,>) direction vector. If no such statements are found then the Heads and Ends of the two loops are interchanged.

```
TYPE
   Stmt: Si, Sj, Sl;

PRECOND
  Code_Pattern                    /* Find a constant definition */
    any Si: Si.opc == assign AND type(Si.opr_2) == const;

  Depend                          /* Use of Si with no other definitions */
    any (Sj,pos): flow_dep(Si, Sj,(=));
    no (Sl,pos): flow_dep(Sl, Sj,(=)) AND (Si != Sl)
            AND operand(Sj,pos) != operand(Sl,pos);

ACTION                            /* Change use of Si in Sj to be constant*/
  modify(operand (Sj,pos), Si.opr_2);
```

Figure 1. GOSpeL specification of Constant Propagation

```
TYPE
  Stmt: Sn, Sm;
  Tight Loops: (L1, L2);

PRECOND
  Code_Pattern                    /* Find two nested loops */
   any(L1, L2);

  Depend                          /* Ensure invariant loop headers */
                                  /* No flow_dep statement and direction of (<,>) */
   no L1.head: flow_dep(L1.head, L2.head)
   no Sm, Sn : mem(Sm, L2) AND mem(Sn, L2),
         flow_dep(Sn, Sm, (<,>));

ACTION                            /* Interchange heads and tails */
  move(L1.Head, L2.Head);
  move(L1.End, L2.End.prev);
```

Figure 2. GOSpeL specification of Loop Interchange

2.1. Declaration Section

Variables are defined to be one of the following types: **Statement**, **Loop**, **Nested Loops**, **Tight Loops**, or **Adjacent Loops**. The domain of these types are the particular code elements in the intermediate code representation. A variable of type **Statement** can have as its value any of the intermediate code statements in the program. All types have pre-defined attributes denoting the next or previous code element of that type.

Statement types have pre-defined attributes indicating the first, second and third operand and the operation. The pre-defined attributes of type **loop** include the loop body, which identifies all the statements in the loop, loop control variable, initial value, final value, head of the loop, and end of the loop. **Tight Loops** restrict **Nested Loops** by ensuring that there are no statements between the two loops. Variables receive their values as a result of various operations performed by operators such as **any**, **all**, and **no**.

In the declaration section, variables are defined following the keyword **TYPE** using the format:

type: *id_list* ;

The *id_list* for **Statement** and **Loop** is simply a list, but **Nested Loops**, **Tight Loops**, and **Adjacent Loops** require parenthesized pairs of identifiers.

2.2. Pre-condition Section

In order to specify a code transformation and conditions under which it can be applied safely, the pattern of code (e.g., constant operands) and the data and control dependence conditions (if any) that are needed must be given. These two similar components constitute the precondition section of a specification. The keyword **PRECOND** is followed by the keyword **Code_Pattern**, which precedes the code pattern specifications and **Depend** which precedes the dependence specification, forcing the user to order the code pattern specifications prior to the dependence specifications. This ordering is enforced for ease of converting the specifications to executable code.

The code pattern section specifies the format needed for the statements and loops involved in the optimization. The code pattern specification consists of a quantifier followed by the elements needed and the required format of the elements.

quantifier element_list : *format_of_elements*;

The quantifier operators **any** and **all** return an element or all the elements, respectively, of the requested types if a match is successful. The **no** operator returns null and warns the user that no statement has been specified for pattern matching. The second part of the code pattern specification describes the format of the type of elements required. If **Statement** is the element type, then the format typically restricts the statement's operands and operator. Thus, if constants are required as operands or if loops are required to start at iteration 1, this requirement is specified in the *format_of_elements*. Expressions can be constructed in *format_of_elements* using the **AND** and **OR** conjunctives with their usual meaning.

The second component of the precondition section is the specification of data or control dependences that are required. The dependences are specified using the names of the code elements listed in *element_list*. The dependence specification consists of expressions that return a boolean value and the set of elements that meet the conditions. The general form of the dependence specification is

element_quantifier element:
sets_of_elements, *dependence_conditions*;

The description of the *sets_of_elements* is specified before the dependence conditions only for ease in automatically converting the specifications to executable code.

The element quantifier may be **any**, **no**, or **all**. **Any** specifies a condition for one statement, **all** specifies all statements that satisfy the condition, and **no** specifies that no statement exists with the condition. The dependence condition returns two objects: the collected set and a truth value. Optionally, the user may request the position of the dependence within the statement (Sj,pos) to also be returned for each element in the collected set, as shown in Figure 1.

The *sets_of_elements* component permits the author of the specification to define set membership of elements. The **mem***(Element, Set)* operation specifies that *Element* is a member of the defined *Set*. *Set* may be described using predefined sets, the name of a specific set, or an expression involving set operations and set functions. An example of a predefined set is **path (ID, ID')** that has as its values the set of statements along a path designated by ID and ID'. The *sets_of_elements* items may be separated by an **AND** or **OR** operator.

The *dependence_conditions* describe the data and control dependences of the code elements and take the form:

type_of_dependence(*StmtId*, *StmtId*, *Direction*);

The dependence type can be either flow dependent (**flow_dep**), anti-dependent (**anti_dep**), output dependent (**out_dep**), or control dependent (**ctrl_dep**). *Direction* is a description of the direction vector, where each element of the vector consists of either a forward, backward or equivalent direction (represented with <, >, =, respectively), or **any** which allows any of the three directions. This representation is needed to specify loop-carried dependences of array elements for parallelizing transformations. This direction vector may be omitted if loop-carried dependences are not relevant.

Semantically, the dependence specification has two roles. First of all, variables are assigned values using **any**, **all**, and **no**. The dependence conditions place further restrictions on the components of the computed sets. Secondly, the list of dependence specifications are evaluated for a truth value. If all dependence specifications and code pattern specifications are true, then the precondition evaluates to true.

As an example, the following specification is for one element named Si that is an element of Loop 1 such that there is a Sj, an element of Loop 2, and there is either a flow dependence or an anti-dependence between Si and Sj.

Depend
any Si: mem(Si, L1) AND mem(Sj, L2),
flow_dep(Si, Sj,(=)) OR anti_dep(Si, Sj, (=));

2.3. Action Section

The actions of applying transformations can be decomposed into a sequence of the following five primitive operations. The semantics of each are indicated below. These operations are overloaded in that they can apply to different types of code elements. In the following descriptions, *a*, *b* and *c* refer to any type of code element. The five actions are:

Delete (*a*): delete *a*.

Copy (*a*, *b*, *c*): copy *a*, place it following *b*, and name it *c*.

Move (*a*, *b*): remove *a* from its original position and place it following *b*.

Add(*a*, Element_description, *b*): add an element described by Element_description, place it following *a*, and call it *b*.

Modify (Operand(S,i), New_operand): modify Operand i of statement S to be New_operand.

These actions are combined to fully describe the optimization. It may be necessary to repeat some actions for all statements found in the precondition. Hence, a list of actions may be preceded by **forall** and an expression describing the elements to which the actions should be applied.

The flow of control in a specification is implicit with the exception of the **forall** construct available in the action section. In other words, the **ACTION** keyword acts as a guard that does not permit entrance into this section unless all conditions have been met.

The initial design of the GOSpeL language was fine-tuned by having other researchers, some very familiar with optimizations and some not, write optimizations in GOSpeL. These users were able to specify known optimizations without any help. One of the changes suggested by these users was to change an original quantifier **one** to **any** for ease of understanding.

3. Description of GENesis

The GENesis tool analyzes a GOSpeL specification and generates C code to perform the appropriate pattern matching, check for the required data dependences, and call the necessary primitive routines to apply the optimization. Figure 3 presents a pictorial view of GENesis and its use. A source program that is to be optimized is converted to an intermediate representation (usually as part of the compilation process) and data dependences are computed. The intermediate code and the data dependences are input into the generated optimizer (OPT), and optimized intermediate code is produced.

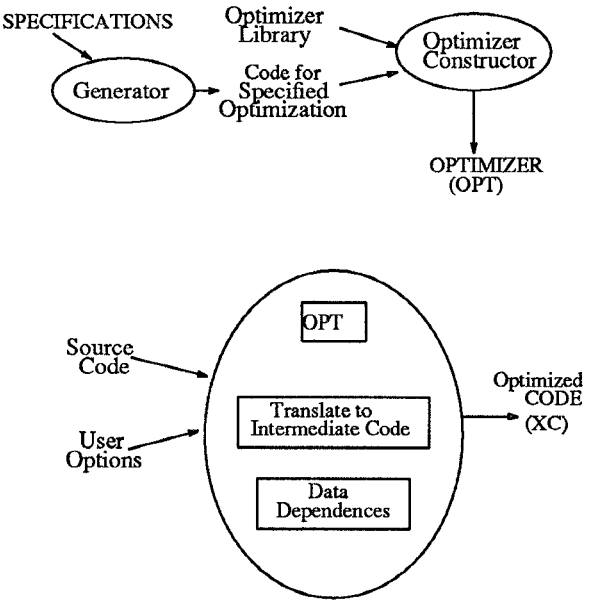

Figure 3. Overview of GENesis

There are three parts to GENesis: a generator, a library, and a constructor. The generator produces code for the specified optimizations, which utilize the predefined routines in the optimizer library. The constructor packages all of the produced code and the library routines within an interface, which prompts interaction with the user.

GENesis analyzes the GOSpeL specifications using LEX and YACC, producing the data structures and code for each of the three sections of a GOSpeL specification. The generator producer first establishes the data structures for the code elements in the specifications. Code is then generated to find elements of the required format in the intermediate code. Code to verify the required data dependences is next generated. Finally, code is generated for the action statements. The algorithm used in GENesis is given in Figure 4.

Step 1:
Input the GOSpeL specifications to GENesis

Step 2:
Analyze the GOSpeL specifications using LEX and YACC and generate code to:

a. set up the data structures defined in the type section

b. search for the patterns specified in code_pattern section - call the necessary pattern matching routines to find the specified types (e.g., find_nested_loops, find_statement)

c. check data dependences for those elements specified in the precondition section of Depend

d. perform actions by calling pre-defined library routines for primitive actions

Step 3:
Construct the optimizer by:

a. Packaging the produced code for all optimizations and library routines

b. Creating the interface from a template to:

i. Read the source code

ii. Convert source to intermediate representation

iii. Allow interaction with the user

1. Select optimization(s) to perform

2. Select application points

3. Override dependence restrictions

iv. Compute the data dependences

v. Perform the optimization at user's request

vi. Return to iii. until user quits session

Figure 4. The GENesis Algorithm

The generated code relies on a set of predefined routines found in the optimizer library. These routines are optimization independent and only represent routines typically needed to perform optimizations. The library contains pattern matching routines, data dependence verification procedures, and code transformation routines. The pattern matching routines search for loops and statements. Once a possible pattern is found, the generated code is called for verification of such items as operands, opcodes, initial and final values of loop control variables.

When a possible application point is found in the intermediate code, the data dependences must be verified. Data dependence verification may include a check for the non-existence of a particular data dependence, a search for all dependences, or a search for one dependence within a loop or set. The generated code may simply be an "if" to ensure a dependence does not exist or may be a more complex conglomeration of tests and loops. For example, if all statements dependent on "Si" need to be examined, then code is generated to collect the statements. The required direction vectors associated with each dependence in the specification are matched against the direction vectors of the dependences that exist in the source program.

If the dependences are verified then the action is executed. Routines consisting of the actions specified in the ACTION section of the specification are generated for the appropriate code elements.

The constructor compiles the optimizer library and the generated code to produce the optimizer (OPT). The constructor also generates an interface to execute the various optimizations. The interface to an optimizer reads the source code, generates the intermediate code and computes the data dependences. The interface also queries the user for interactive options. This interactive capability permits the user to execute any number of optimizations in any order. The user may elect to perform an optimization at one application point (possibly overriding dependence constraints) or at all possible points in the program. The interface permits the user to decide if the data dependences should be re-calculated between execution of each optimization.

3.1. Prototype Implementation

In order to test the robustness of the GENesis system, a prototype implementation was developed. The prototype was used to generate optimizers for a number of optimizations. For any optimization specified (e.g., xxx), the generator produces four procedures: set_up_xxx, match_xxx, pre_xxx, and act_xxx. These procedures correspond to the sections in the specifications.

In our implementation, an optimizer consists of a driver that calls the routines that have been generated specifically for that optimizer. The format of the driver is the same for any optimizer generated. The driver calls procedures in the generated call interface for the specific optimization. The call interface in turn calls the generated procedures that implement the optimization. The standard driver is given in Figure 5 using pseudocode. Notice that the driver calls four procedures (*set_up_OPT*, *match_OPT*, *pre_OPT*, and *act_OPT*) that are found in the call interface for the specific optimization. The call interface code simply calls the generated optimization specific code. In other words for CTP, the *set_up_OPT* procedure consists of a single call to *set_up_CTP*. The driver requires a successful pattern match from *match_CTP* and *pre_CTP* in order to continue. Thus, the *match_OPT* and *pre_OPT* of the call interface procedures return a boolean value. The C code that was actually gen-

erated to implement the pattern matching, dependence checking, and actions for CTP is given in Figure 6.

```
Driver
Call set_up_OPT to initialize stlp structure.
pat_suc := True
WHILE (pat_suc AND NOT Done) DO
  pat_suc := match_elements(stlp)
  IF (pat_suc) THEN DO
    match_suc := match_OPT
    IF (match_suc) THEN DO
      pre_suc := pre_OPT
      IF (pre_suc) THEN DO
        act_opt
        Done := True;
      ENDIF
    ENDIF
  ENDIF
ENDWHILE
END
```

Figure 5. The Driver Algorithm

The generated *set_up* procedure consists of code that initializes data structures for each element specified using **any** or **all** in the **PRECOND** section. The *stlp* data structure contains identifying information about each statement or loop variable specified in the **TYPE** section. For type **Statement**, an entry is initialized with the type and corresponding identifier. If a loop type variable is specified, additional flags for nested or adjacent loops are set in the *stlp* entry. These entries are filled in as the information relevant to the element is found. For the CTP example, an *stlp* entry is initialized to type "Statement" and identifier "Si" when the optimizer executes procedure *set_up_CTP*, for a search for this statement is required initially.

After the *set_up_OPT* procedure terminates, the driver initiates the search for the statement recorded in the *stlp* table by calling *match_OPT*.

In the example, the driver would call *match_CTP*. In the *match_CTP* procedure, the pattern matching routine *find* searches the intermediate code for a quad (statement "Si") that has opcode of "ASSGN" and a constant operand. If the source program's statement does not match, then the optimizer driver re-starts the search for a new statement.

```
set_up_CTP(){
 /* set up stlp for one element:statement, Si */
stlp[1].kind = Statement;
strcpy (stlp[1].desc.stmt.id, "Si");
num_elems = 1;
return(1);
}
match_CTP() {
if (quad[find("Si",0)].opc.kind != ASSGN)
   return(0);  /* if quad's opcode isn't ASSGN, fail */
if (quad[find("Si",0)].opra.kind != const)
   return(0);  /* if quad's operand_a isn't constant, fail */
return(1);  /* match was successful for Si */
}
pre_CTP() {
ins_stmt(-1, "Sj"); /* insert: Statement, Sj */
/*If flow dependent Sj exists, assign its quad number */
if((stlp[num_elems].desc.stmt.stmt_num =
    dep(LST,FLOW,find("Si",0),0,0,1,EQ)));
else return(-1);
ins_stmt(-1, "Sl"); /* insert: Statement, Sl */
/*If suitable Sl exists, assign its quad number */
while((stlp[num_elems].desc.stmt.stmt_num =
    dep(LST,FLOW,0,find("Sj",0),0,1,EQ)))
    /*compare quad_numbers and operand involved in
    dependence */
    if (find("Si",0)!=find("Sl",0) &&
    dep_opr(find("Sj",0))==dep_opr(find("Sl",0)))
    return(-1);
return(1);
}
act_CTP() { /* modify one of quad Sj's operands */
/* repl compares AND replaces operand */
/* involved in dependence if it matches */
modify(&quad[find("Sj",0)].opra,
repl(&quad[find("Si",0)].oprc,&quad[find("Sj",0)].opra,
&quad[find("Si",0)].opra,&quad[find("Sj",0)].opra),-1);
modify(&quad[find("Sj",0)].oprb,
repl(&quad[find("Si",0)].oprc,&quad[find("Sj",0)].oprb,
&quad[find("Si",0)].opra,&quad[find("Sj",0)].oprb),-1);
return(1);
}
```

Figure 6. The Generated Code for CTP

```
Function dep;
Input:
  1. TYPE of search - LST or IF
  2. KIND of dependence (anti, flow, output, ctrl)
  3. Statements involved
     TYPE == IF: both starting and terminating statements of dependence
     TYPE == LST: either the starting or terminating statement
  4. FLG signaling the number of dep call
  5. NUMber of elements in direction vector
  6. DIRection vector to be matched

Output:
  0 - no dependence found
  1 - dependence for IF found
  value - statement number of dependence

  Si = emanating quad number
  Sj = terminating quad number
  if (TYPE = IF) then begin
    if (dependence from Si to Sj = KIND)
      and (DIR matches)
      return(1);
    else
      return(0);
  else begin
    if (Si known) then begin
      Sj = first terminating statement with
       ( dependence = KIND) and (DIR matches);
      save[flg] = Sj;
      return(Sj);
    else begin
      Si = first emanating statement with
       (dependence = KIND) and (DIR matches);
      save[flg] = Si;
      return(Si);
    endif;
  endif;
end dep.
```

Figure 7. The *dep* algorithm

The next routine called is *pre_OPT* to check for data dependences. For CTP, the *pre_CTP* procedure inserts an element into the *stlp* structure for each dependence condition statement. Sj is inserted into *stlp* and the procedure *dep*, given in Figure 7, is called to find the first statement that is flow dependent on Si. If one is not found then the condition fails. Sl is inserted and the procedure *dep* is called again. Each Sl such that Sl is flow dependent on Sj is examined to determine if the operand of Sl causing the dependence is the same variable involved in the dependence from Si to Sj. If such an Sl is found then the condition fails.

The last procedure to be called is *act_OPT*, which translates to *act_CTP* for CTP. Procedure *act_CTP* simply modifies the operand collected in Sj. The call to *repl* compares the first and second parameters. Thus, the first call to *modify* considers "operand a" of Sj for replacement and the second call considers "operand b" for replacement, effectively implementing the pattern matching needed for determining the operand position of a dependence. *act_CTP* is called by the driver only if *match_CTP* and *pre_CTP* have terminated successfully.

There are three modules of C code involved in the generation of an optimizer by GENesis: the generator, the generated code for an optimizer, and the non-optimization specific library. The generator consists of 1,735 lines of code (including LEX and YACC specifications). An optimization consists of 99 lines on the average, where the call interface consists of 29 lines of code, and the four generated procedures consist of 70 lines on the average. The non-optimization specific code in library is 1,873 lines. These lines of code do not include the routines needed to convert the source to intermediate code or the data flow routines.

The existing GENesis prototype can be expanded in various aspects to permit user flexibility. Such implementation expansions include a graphical user interface to guide the user in the application of transformations. The current implementation only permits the user to provide a suggested application point by inputing the intermediate code location. Not all of the features in GOSpeL have been implemented in the prototype; however, the implementation of these features would not pose any problems. Example restrictions include a step by one in loop increments and no expressions are included as code elements in the **forall** construct of the **ACTION** section.

Because of our interests, the optimizers currently implement only one optimization. For a sequence of optimizations to be applied to program code, the various optimizers are called in the desired sequence. However, it is fairly easy to change the implementation to have the driver sequence through a number of optimizations. The data flow analyzer may have to be called after each application.

4. Experimentation

We are performing experiments using optimizers produced by GENesis to determine the application cost and quality of code produced by the optimizers and properties of optimizations. In this section, we discuss some of the results obtained so far. In all, optimizers were produced for ten optimizations including both traditional and parallelizing optimizations. The optimizations are Copy Propagation (CPP), Constant Propagation (CTP), Dead Code Elimination (DCE), Invariant Code Motion (ICM), Loop Interchanging (INX), Loop Circulation (CRC), Bumping (BMP), Parallelization (PAR), Loop Unrolling (LUR), and Loop Fusion (FUS). Experimentation was performed using programs found in the HOMPACK test suite and in a numerical analysis test suite[3]. HOMPACK consists of FORTRAN programs to solve non-linear equations by the homotopy method. The numerical analysis test suite included programs such as the Fast Fourier Transform and solving non-linear equations using Newton's method. A total of ten programs were used in the experimentation.

We first compared the quality of code produced by our optimizers with that produced by hand-crafted optimizers. Our optimizers found the same application points and the resulting code was comparable to that produced by the hand-crafted optimizers. There were no extraneous statements, and the optimizations were correctly performed.

In the test programs, CTP was the most frequently applicable optimization (often enabled) while no application points for ICM were found. It should be noted that the intermediate code did not include address calculations for array accesses, which may introduce opportunities for ICM. CTP was also found to create opportunities to apply a number of other optimizations, which is to be expected. Of the total 97 application points for CTP, 13 of these enabled DCE, 5 enabled CFO and 41 enabled LUR (assuming that constant bounds are needed to unroll the loop). CPP occurred in only two programs and did not create opportunities for further optimization.

To investigate the ordering of optimizations, we considered the optimizations FUS, INX and LUR which have been found to theoretically enable and disable one another[13]. In one program, FUS, INX, and LUR were all applicable and heavily interacted with one another by creating and destroying opportunities for further optimization. For example, applying FUS disabled INX and applying LUR disabled FUS. Different orderings produced different optimized programs. The optimizations also interacted when all three optimizations were applied; when applying only FUS and INX, one instance of FUS in the program destroyed an opportunity to apply INX. However, when LUR was applied before FUS and INX, INX was not disabled. Thus, users should be aware that applying an optimization at some point in the program may prevent another optimization from being applicable. To further complicate the process of determining the most beneficial ordering, different parts of the program responded differently to the orderings. In one segment of the program INX disabled FUS, while in another segment INX enabled FUS. Thus, there is not a "right" order of application. The context of the application point is needed. Using the theoretical results of interactions from the formal specifications of optimizations[13] as a guide, the user may need multiple passes to discover the series of optimizations that would be most fruitful for a given system.

Another set of experiments evaluated the cost and benefit of applying optimizations. The cost of applying an optimization was estimated using the number of checks to determine preconditions and the number of operations to apply the code transformation. As an optimization was actually applied, this value was computed by using code that GENesis produced. These cost values were validated by running the optimizers and timing their execution. We found that the estimated times very closely reflect the actual times. The expected benefit of applying an optimization was computed by estimating the impact the optimization has on execution time, taking into account code that was parallelized and code that was eliminated. Different architectural characteristics were considered, including vectorization and multi-processing. These costs can be used in a number of ways. The costs can be used in determining whether an optimization should be included in a production optimizer. As an example, INX was found to be a relatively inexpensive operation with large benefits. CTP is inexpensive to apply, and it also enables many parallelizing optimizations. FUS was found to apply in only one test case and is a fairly expensive optimization to apply with little expected benefit unless various types of memory hierarchies are part of the parallel system. We have yet to experiment with this type of architectural consideration.

If the cost of an optimization is very high, then alternative methods of specification should be attempted. In applying the optimizations, it was found that different specifications will produce different implementations of the optimization, which have an impact on the cost. For example, if the specification of LUR requires that both the upper and lower limits are constant, LUR is less costly to apply if the upper limit is checked before the lower bound. Our experimentation showed that it is more likely for the upper limit to be variable than the lower limit, thus discarding a non-application point earlier.

The costs were also used to determine a better way to implement optimizations. A number of optimizations involved the determination of membership when checking

for preconditions. Two straightforward ways of implementing the checking are (1) to determine statements that are members and then check for the desired dependences, and (2) to consider the dependences of one statement and check the corresponding dependent statements for membership. We found that the cost of implementing the optimizations using these approaches varies tremendously and is not consistently better for one method over the other. Using heuristics, GENesis was changed to select the least expensive method on a case by case basis. In the tests performed, we found that the heuristic correctly selected the best implementation.

5. Related Work

Although numerous optimizing compilers and optimization systems, such as Parafrase-2[12], ParaScope[2], and PTRAN[1] have been designed and developed, this research focuses on the *automatic* generation of optimization systems. Techniques for the automatic generation of various peephole optimizers have been reported[4-6,8]. These optimizers apply localized optimizations found by pattern matching on assembly or machine-level code. They have no facilities for handling global information, which is needed to perform the optimizations of interest in this research. GENesis works at a higher level program representation and can handle various types of program code structures. It also incorporates global information in the form of data and control dependences. However, it should be noted that GENesis could also be used to produce peephole optimizers.

A recently developed technique presents a language for specifying optimizations on assembly language and an implementation of the language in Prolog[7]. There are no provisions for incorporating global information, although some simple data flow analysis is performed. The machine-level nature of the implementation does not allow for easy recognition of array structures or source-level constructs such as tightly nested loops.

There are other optimization systems with goals that differ from those in this research, which is to optimize program code. Some of these optimization systems attempt to optimize high-level specifications into more concrete specifications using various optimizations and the user's help. Partsch and Steinbruggen[11] give an excellent overview of these general optimization systems.

The need for an optimization system permitting the specification of a sequence of transformations, and the automatic generation of the transformations, has long been recognized[9]. Such a system enables a user to create and easily implement novel optimizations which may be of particular benefit to the system in hand. GENesis is such an optimization system. It permits the designer to create and specify the optimizations and control the application order. In addition, GENesis uses a well defined specification technique for specifying optimizations.

Acknowledgement
The authors thank Christopher Fraser for his many helpful comments and suggestions on this work.

APPENDIX -- BNF for Depend Section

```
Pre    ⇒ quant stmtlst : Elemlst condlst ;
quant  ⇒ ANY | NO | ALL
Elemlst ⇒ mem ( ID , setexp ) Elmore | ε
Elmore ⇒ , | AND Elemlst | OR Elemlst
setexp ⇒ stxp | comp ( stxp , setexp )
stxp   ⇒ ID | PATH ( ID , ID )
mem    ⇒ MEM | NMEM
comp   ⇒ INTER | UNION
condlst ⇒ clist | NOT ( clist )
clist  ⇒ term | condlst OR term
term   ⇒ conds | term AND conds
conds  ⇒ type ( StmtId , StmtId direct ) | ( condlst )
         | ( StmtId relop StmtId )
         | op_fn relop op_fn
op_fn  ⇒ operand ( StmtId )
type   ⇒ FLOW | OUT | CTRL | ANTI
direct ⇒ , ( sub more ) | ε
sub    ⇒ relop | ANY | ε
more   ⇒ , sub more | ε
relop  ⇒ < | > | = | <= | >= | != | ==
StmtId ⇒ ID cont | (ID , POS )
cont   ⇒ . NXT cont | . PREV cont | . HEAD cont
         | . END cont | . LABEL | . FINAL | . INIT
         | . BODY | . LCV | ε
```

References

1. F. E. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hseh, and V. Sarkar, "A Framework for Determining Useful Parallelism," *Proceedings of the 1988 International Conference on Supercomputing*, pp. 207-215, St. Malo, France, February, 1988.
2. Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley, and Jaspal Subhlok, "The ParaScope Editor: An Interactive Parallel Programming Tool," *Proceedings of Supercomputing '89*, pp. 540-549, Reno, Nevada.
3. Richard Burden and J. Douglas Faires, in *Numerical Analysis*, Prindle, Weber & Schmidt, Boston, MA, 1989.