SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(3), 307–325 (MARCH 1994)

The Design and Implementation of Genesis

deborah whitfield
Department of Computer Science, Slippery Rock University, 106 Maltby Center, Slippery Rock, PA 16057-1326, U.S.A. (email: dlw@sruvm.sru.edu)

and

mary lou soffa
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A. (email: soffa@cs.pitt.edu)

SUMMARY

Although code optimizations are necessary to parallelize code, few guidelines exist for determining when and where to apply optimizations to produce the most efficient code. The order of applying optimizations can also have an impact on the efficiency of the final target code. However, determining the appropriate optimizations is difficult due to the complex interactions among the optimizations, scheduler and architecture. To aid in selecting appropriate optimizations, an optimizer generator (Genesis) is presented that produces an optimizer from specifications of optimizations. This paper describes the design and implementation of Genesis and demonstrates how such a generator could be used by optimizer designers. Some experiences with the generator are also described.

key words: Parallelizing compiler optimizations Automatic generation Transformations

INTRODUCTION

Traditionally, code optimizations were applied to improve the space and time performance of executable code. With the advent of numerous parallel architectures, deciding what optimizations* to apply and the order of their application became quite complex. The performance of the parallelized code is very much dependent on both the underlying architecture and the scheduler that is used to schedule the parallel events on the processors. For some architectures and schedulers, applying particular optimizations would, in fact, degrade the performance on the parallel system. Also, the order of application can have a significant impact on the performance of the optimized code. Owing to the dependence of optimizations on the architecture, it may be beneficial to design or tailor the optimizations to match a particular architecture. Since there does not exist a formal foundation that aids in deciding which optimizations to apply and where, experimentation is essential to determine the properties and value of the optimizations on program code.

One experimental approach to determine the most appropriate optimizations and

* The term optimization encompasses both traditional optimizations and optimizations used for parallelization.

CCC 0038–0644/94/030307–19 Received 8 September 1992
Revised 29 September 1993

their application order is to implement a number of optimizations in an optimizer, perform the optimizations on the code, and then evaluate the performance of the optimized code on the parallel system that is to be used. However, actually implementing an optimizer is a very time consuming process, especially when the detection of complex conditions and global control and data dependency information are required. Because of the *ad hoc* manner in which such an optimizer is usually developed, the addition of other optimizations or even the deletion of optimizations may require a substantial effort to change the optimizer. In some cases, the entire optimizer may have to be rewritten. The development of optimization systems such as Parafrase-2,[1] Parascope,[2] and PTRAN[3] used this approach.

Although automatic code generation has been used in the development of peephole optimizers,[4–7] it has yet to be exploited for high-level optimizations requiring global dependency information, such as needed for parallelizing optimizations. In this paper, an approach to producing optimizers is developed that entails users specifying the desired optimizations in a specification language, the *G*eneral *O*ptimization *Spe*cification *L*anguage (Gospel). These specifications are then used to automatically generate an optimizer that can be used in experimentation or in a production setting.[8] This paper focuses on the prototype implementation of an automatic optimizer *gen*erator (Genesis) that uses optimizations specified in Gospel. Thus, this approach extends the automatic generation of lexical analyzers (e.g. LEX) and parsers (e.g. YACC) to include the automatic generation of optimizers (e.g. Genesis).

Overview

The generation of an optimizer by Genesis requires the specification of those optimizations to be included in the optimizer. In this work, parallelizing optimizations are specified in the Gospel language and are input to the generator, Genesis. The output from Genesis is executable code that implements the optimizations. For generality, Genesis produces an optimizer for a high-level intermediate representation of the source program that maintains loop control structures and array references. Thus, the system is source code independent and can be used for any language that can be represented by the intermediate code (e.g. FORTRAN, Pascal, C). The design of Genesis allows the user to specify whether the optimization should be applied at all valid program points or should be applied under user direction. The higher level of the intermediate code (i.e. containing control structures) allows the user to interact at the source level for loop optimizations that are typically applied for parallel systems.

The value of such a tool is that it allows

1. the experimental investigation of the performance of the optimizations on the system under consideration. The performance of an optimization may influence its inclusion in the final optimizer.
2. an investigation into which optimizations should be included for a given configuration of architecture and scheduler. Experimentation has shown that some optimizations can be a detriment to some schedulers.[9]
3. an investigation into the order in which optimizations should be applied. An examination of the interactions between optimizations may alter the order in which the optimizer applies the selected optimizations.
4. the development of optimizations that are particularly targeted to the architecture

and/or scheduler. Optimizations may be developed to exploit a particular architecture or scheduler.

5. the investigation of the cost versus the benefit of optimizations. The cost and expected benefit of an optimization may be used to determine the merit of including the particular optimization in a production optimizer.

Scenario using Genesis

Consider a scenario where a user is considering three parallelizing optimizations to apply to parallelize a program segment. Knowing that optimizations can interact and that determining an optimal order of application is unlikely, the user performs experiments using Genesis to decide the best order in which to apply the three optimizations. The user specifies the three optimizations in Gospel and inputs them to Genesis. Genesis produces three separate optimizers (one for each optimization) so that experimentation may be performed. Alternatively, one optimizer could be produced containing all three optimizations.

For example, consider the application of loop fusion (FUS), loop unrolling (LUR), and loop interchanging (INX) and assume that the more optimizations that are applied, the better the resultant code. Experimentation finds that applying LUR creates additional opportunities for applying INX and applying INX creates opportunities to apply FUS. Also, an application of FUS destroys opportunities for applying INX. Using the previous assumption, this experimentation determines that the ordering ⟨LUR, INX, FUS, LUR, INX, FUS, …⟩ results in code that contains the most optimizations.

At this point in the scenario the user wants to examine the cost of applying the optimizations under consideration. Genesis is used to indicate the relative cost of applying transformations. Based on these costs, the user decides whether or not to incorporate particular optimizations in the final optimizer. Further experimentation may be carried out on a particular machine to determine the benefit of the optimization versus the cost of applying the optimization.

THE DESIGN OF GOSPEL AND GENESIS

This section overviews the specification language, Gospel and its implementation through Genesis. The reader is referred to previous work for more details about Gospel.[8] The power of Gospel is indicated by its use in specifying the 21 optimizations that appear in Appendix I.

The general optimization specification language

Gospel uses a set of constructs common to optimizations to develop well-formed specifications. The overall format of a Gospel specification consists of a name, a type section, a precondition section, and an action section. The format of a Gospel specification is

```
Name
    TYPE
    PRECOND
    Code_Pattern
    Depend
  ACTION
```

The Name of a specification associates a name with an optimization. The conversion of the specification to executable code requires a unique identifier for each optimization to be included in the generated optimizer. The TYPE section declares the types of the required code elements, which can be statements or loops, where loops can be adjacent, nested, or tightly nested. The PRECOND section is subdivided into the Code_Pattern and Depend sections. The Code_Pattern components express the patterns in the code that are acceptable for the application of the optimization. Such patterns permit verification of particular operands and opcodes in the intermediate code statements. In the Depend section, the specification involves the description of statement and operand dependences, direction vectors, and any necessary set membership qualification. The ACTION section specifies the primitive code transformations that combine to perform an optimization.

Gospel is source-language independent, but the prototype assumes an intermediate level representation of the form:

$$\text{Operand}_3 := \text{Operand}_1 \text{ Opcode Operand}_2$$

where Operand$_1$ and Opcode are optional. Appendix II contains the grammar expressing the precondition for the prototype implementation of Gospel.

The example specifications of loop fusion (FUS) in Figure 1 and of constant

```
FUS
TYPE
      Stmt Sn, Sm, Si, Sj, Sk;
      Adjacent Loops: (L1, L2);

PRECOND
 Code_Pattern                 { find adjacent loops with equivalent heads }
      any L1, L2: L1.initial == L2.initial
            AND L1.final == L2.final
            AND L1.lcv == L2.lcv
 Depend                       { no dependence with > first and no definitions reaching in }
      no Sn, Sm: mem (Sn, L1) AND mem (Sm, L2),
            flow_dep (Sn, Sm, (=*, >, any)) OR out_dep (Sn, Sm, (=*, >, any)) OR
            anti_dep (Sn, Sm, (=*, >, any));
      no Si, Sj: mem (Sj, L1) AND mem (Sk, L2),
            flow_dep (Si, Sj, (any)) AND anti_dep (Sj, Sk, (any)) AND (Si != Sk);

ACTION                        { Fuse the loops }
      modify (L1.Head.opr1, L2.Head.label);
      modify (L2.End.opr1, L1.End.label);
      delete (L1.End);
      delete (L2.Head);
```

Figure 1. Gospel specification of loop fusion

folding (CFO) in Figure 2 use the above format. Loop fusion combines two adjacent loops when both iteration structures are equivalent. Such a transformation is used to aid in the parallelization of sequential code.

The specification of FUS in Figure 1 declares five statements (S_n, S_m, S_i, S_j, and S_k) along with two adjacent loops (L_1 and L_2). The Code_Pattern simply requires the existence of any two adjacent loops that have the same initial value, final value, and loop control variable, where any is a Gospel quantifier. The Depend section uses the no quantifier to ensure that there is not a dependence (flow, anti, or output) between the two loops with a backward direction (backward loop dependences are denoted with a $>$, forward with a $<$, and same iteration with a $=$). The second dependency check also uses the no quantifier to ensure that there are no definitions that reach into the loop. The ACTION specification fuses the two loops by modifying the iteration structure and then removing the extraneous loop header and end. Note that this specification requires the loop control variables to be the same, so it is not necessary for this specification to modify the statements within the loop.

The specification of CFO in Figure 2 declares statement S_1. The Code_Pattern checks for a statement that operates on two constants by verifying that the operands are constant and the operation is not assignment. The Depend is empty, as no dependence checks are necessary for CFO. The ACTION specification folds the expression by performing the operation and replacing the second operand with the result and modifying the operation of the statement to be an assignment statement.

Design of Genesis

Genesis analyzes Gospel specifications of optimizations and generates program code to perform the appropriate pattern matching, check for the required data dependences, and call the necessary primitive routines to apply an optimization. Genesis produces an optimizer (OPT) that requires an extended intermediate representation of a program and computed data dependences. The intermediate program code and the data dependences are input to OPT along with any user options, and optimized intermediate code is produced, as depicted in Figure 3. Genesis can produce an optimizer that performs one optimization or multiple optimizations.

```
CFO
TYPE
      Stmt: Si;

PRECOND
 Code_Pattern                       { Find a constant expression }
      any Si: type(Si.opr2) == const
            AND type(Si.opr3) == const
            AND Si.opc != assign;
 Depend                             { No dependence checks }

ACTION                              { Fold the constants into an expression }
      modify (Si.opr2, eval(Si.opr2, Si.opc, Si.opr3);
      modify (Si.opc, assign);
```

Figure 2. Gospel specification of constant folding

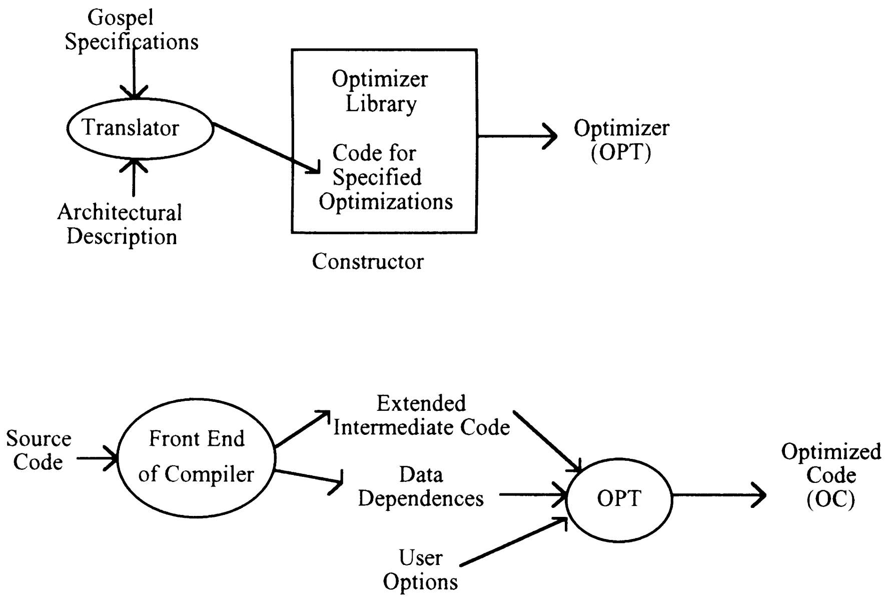

Figure 3. Overview of Genesis

There are three components in the Genesis design; a translator, a library, and a constructor. The translator analyzes the Gospel specifications and then generates the data structures and code for each of the three sections of a Gospel specification, producing optimizations-specific code. The generation and execution of this code are discussed in the next section.

The generated code relies on a set of predefined routines found in the Genesis library. These routines are optimization independent and implement actions typically needed to perform optimizations. The library contains pattern matching routines, data dependence verification procedures, and list manipulation routines. The pattern matching routines search for code elements such as loops and statements. Once a pattern is found, the pattern matching routines can verify other required syntactic items such as operands, opcodes, and initial and final values of loop control variables.

The constructor compiles the library routines and the generated code to produce an 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 permitting 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 recalculated between execution of each optimization.

PROTOTYPE IMPLEMENTATION OF GENESIS

A prototype implementation of Genesis was written to generate C code that implements a specified optimization. Genesis begins its work by parsing each section of the Gospel specification (recall Figures 1 and 2). A generated optimizer consists of a standard driver that calls four procedures specific to the optimization: a set_up procedure initializes data structures required for the specific optimization as a result of parsing the Code_Pattern section of the specification, a match procedure consists of code that searches for patterns specified in the Code_Pattern section of the specification, a pre_condition procedure consists of code that implements the checks requested in the specified Depend section, and an action procedure that implements the code modifications specified in the ACTION section. The generation of an optimizer is described separately from the execution of an optimizer in the following sections.

Description of the generator

The implementation of the generator consists of translating Gospel specifications into C code using LEX and YACC for lexical and syntactical analysis, respectively. The translation of the specifications generates four optimization-specific procedures and is described algorithmically in Figure 4. The algorithm depicts the translation of the four parts of the specification: TYPE, Code_Pattern, Depend, and ACTION.

The TYPE section translation enters the appropriate type (statement, loop, nested

```
Trans
{ Translate TYPE section }
    FOR each Statement, $S_i$ DO           { Enter each declared statement into TransTbl}
            Enter_TransTbl (Statement, $S_i$);
    FOR each Loop, $L_n$ DO                { Enter each declared loop into TransTbl }
            Enter_TransTbl (Loop, $L_n$);

{Translate PRECOND }
    { Translate Code_Pattern sub-section }
            FOR each any element_list: patterns clause DO
                    Generate_Search_For (element_list);
                    Generate_IFs_To_Match (patterns);
            FOR each all element_list: patterns clause DO
                    Generate_Search_FOR_All (element_list);
                    Generate_IFs_To_Match (patterns);

    { Translate Depend sub-section }
            FOR each quantified dependence clause DO
                    CASE operator
                            any: Generate_Loop(dependence, one);
                            all: Generate_Loop(dependence, all);
                            no: Generate_IF (dependence, not);
                            mem: Generate_Loop (relations);
                    ENDCASE

{ Translate ACTION section }
    FOR each action DO                     { Add, Delete, Modify, Copy, Move }
        Generate_action (TransTbl_id, Stmt_components, Code_Loc);
    FOR each forall DO
        Generate_Loop (Loop_Control_description);
```

Figure 4. Translation algorithm

or adjacent loops) for each declared variable into a translation table (TransTbl). The Code_Pattern translation generates the appropriate control structure from the any and all quantifiers to pattern match for the specified elements. Translation of the Depend section results in the generation of the appropriate control structures for satisfying the specified dependences. Translation of the ACTION section produces code to perform each of the actions (add, delete, modify, move, and copy) and possibly repeat the actions.

The generation of the C code introduced a number of implementation problems. The data structures needed for translation and the translation of the specification sections are discussed in this section; the execution of the generated code and structures needed for execution are discussed later.

Translation table data structure

In order to generate code that implements the code elements found in Gospel specifications, a data structure, TransTbl (translation table) is created and used by the generator. A pictorial representation of TransTbl is given in Table I.

This table contains the defining parts of the statement and loop code elements only, as other structures typically found in code can be defined using these two code elements. For a code element, an indication that the element is used with an all quantifier is needed, so all code elements with the specified restrictions are collected for verification. A loop element may additionally require a flag to specify that it is an adjacent or nested loop. Finally, information about loop constants and induction variables may be required for some optimizations. Table I illustrates four of the entries that exist for FUS as specified in Figure 1.

Translation of a Gospel specification

TYPE *section translation.* Translation of the TYPE section results in the creation of the TransTbl as described above. The TransTbl contains an entry for each declared variable of the TYPE section. When processing the precondition section, the quantifier flag is set in the table.

PRECOND *section translation.* Translation of the Gospel Code_Pattern specification in the PRECOND section includes the generation of code to create an OptTyp table needed when the generated optimizer executes. The OptTyp table contains an entry for each element of the TransTbl that has a non-null quantifier, thus recreating the

Table I. TransTbl

Character identifier	Quantifier flag	Statement or loop flag	Loop information
S_n	No	Statement	
S_m	No	Statement	
S_k	Null	Statement	
L_1	Any	Loop	Adjacent
L_2	Null	Loop	Adjacent

elements of the TransTbl that require further verification. The code to create OptTyp table entries is generated when Code_Pattern statements that begin with all or any are encountered. For example, suppose the Gospel specification includes

```
all Stmt_1: patterns
```

This results in code to create a new set for the statements that match patterns when the optimizer is executed. The patterns part of the specification results in the generation of a match procedure that consists of conditions that verify code elements.

The translation of the Gospel Depend section of a specification is controlled by the any, all, and no quantifiers in the specification. These operations dictate the major control structures of the generated pre_condition procedure and the dependence conditions become conditions in the generated optimizer.

The code generated for each of the quantifiers varies. For an any quantifier, a loop is generated to search for the next single occurrence of the specified conditions. For an all quantifier, a loop is generated to collect all of the specified intermediate code statements that meet the condition. A no quantifier signals the generation of code identical to the code generated for an all quantifier, except that an IF statement is also generated to ensure that the reverse condition exists. Additionally, a specified mem operator results in the generation of a loop that is used to determine whether a variable is a member of the specified set. (The set specification may include union and intersection expressions). An example of the code generated for the all operation is given later along with an explanation of its execution.

ACTION *section translation.* Each of the specified actions in the Gospel ACTION section is directly translated into a procedure call within the generated action procedure. There are two exceptions from this direct translation. First, some specified code elements are greater than one code unit in size (e.g. the pre-header of a loop). When such objects are identified, a loop is generated to perform the same action on the entire object. Secondly, forall Stmts_in_expression may precede the specified actions. When such a Gospel construct is found, a WHILE loop is generated to manipulate all of the elements referred to by Stmts_in_expression.

Description of the optimizer

An optimizer produced by the Genesis prototype consists of the generated optimization-specific code and non-optimization-specific code that is required by all optimizations (i.e. library routines). These parts are assembled by a constructor. The execution of the generated optimizer is explained next.

The main procedure of the generated optimizer is a driver that controls the execution of the optimization specific code. The format of the driver is the same for any optimizer generated in that the driver calls on the four optimization-specific procedures. However, some optimizers have complex patterns to be matched resulting in the generation of multiple match procedures. Hence, a call interface is used as a link between the driver and the optimization-specific procedures. A high-level description of the standard driver is given in Figure 5. Notice that the driver requires the existence of four optimization-specific procedures and requests the call interface to call the generated optimization-specific code.

```
Driver
set_up();                                    { Initialize OptTyp table }
Done := False;
WHILE (NOT Done) DO
        match_success := match();            { Match the code patterns }
        IF (match_success) THEN
                pre_success := pre_condition();   { Verify the dependences }
                IF (pre_success) THEN
                        action();            { Perform actions of the optimization }
                        Done := True;
                ENDIF
        ENDIF
ENDWHILE
END
```

Figure 5. The driver algorithm

Optimization table data structure

An OptTyp table maintains data about each variable that is qualified in the PRECOND specification. The OptTyp table may not have as many entries as the TransTbl, which contained an entry for each declared variable.

When the optimizer executes, each entry in the OptTyp table requires a field to describe its code location, pointer_to_code_location. For statements, this field is a simple pointer to the intermediate code statement, but loop-typed elements are more complex and require several pointers to locate the loop elements:

```
Loop Header (Initial, Final loop values)
    Loop Body
Loop End
```

The skeleton of the OptTyp table is given in Table II. This table illustrates the possible fields of the specification variables at run time and gives an example of valid entries for the execution of loop fusion. The first four fields are a copy of the TransTbl entries. The last field indicates the intermediate code locations for the loop head, initial value, final value, and start and end of the loop.

Execution of the generated optimizer

Consider a scenario where loop fusion is to be applied to a given source program. The interaction of the Genesis routinues during this scenario is illustrated in Figure 6.

Table II. OptTyp table

Character identifier	Quantifier flag	Statement or loop flag	Static loop information	Location of loop parts
L_1	Any	Loop	Adjacent	7,8,9,11,18
L_2	Any	Loop	Adjacent	19,20,21,23,26

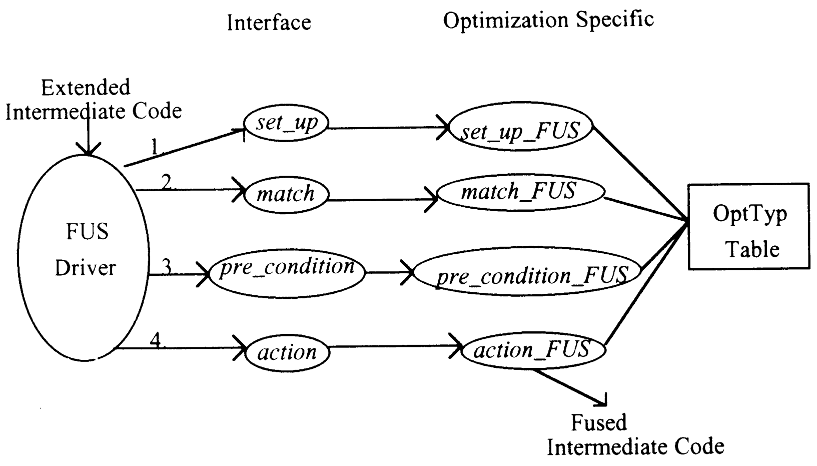

Figure 6. Application of loop fusion

The driver calls set_up within the call interface which in turn calls set_up_FUS to complete the initializations. Next, the driver indirectly calls match_FUS to attempt to satisfy the specified code patterns in the source program. If this attempt fails, then the pattern matcher is restarted from the last code segment attempted; if a pattern is found, the specified dependences are verified. The dependences are verified by pre_condition_FUS which is called by pre_condition from within the call interface. If the verification of the dependences fails, then the pattern matcher is restarted. The search continues until the end of the code is found or all specified conditions are met, at which point the actions of loop fusion are executed by indirectly calling action_FUS. As the execution of set_up, match, and action procedures is obvious, only the execution of pre_condition is described in greater detail.

Typically, the dependency conditions are quantified with any, all, or no. The code executed for these quantifiers varies. However, a generalized description of the all quantifier is given in this section to indicate a typical segment of pre_condition.

The occurrence of all in a Gospel specification results in the generation of code as seen in Figure 7. All statements other than S_i that are flow dependent on S_j are

```
Procedure All_Example     { all S_l: flow_dep(S_l, S_j) AND (S_l != S_i) }
  BEGIN
        OptTyp_look_up (S_l)                    {locate S_l in OptTyp table }
        Set_S_l ⇐new_set(S_l)                   {Create a new set to contain S_l }
        element ⇐ first_dep(flow, (S_l, S_j))   {Find first dependent element }
        WHILE (element)                         {Find all flow dependences }
              if element != S_i                 {S_i is previously determined}
                    Set_S_l ⇐Set_S_l ∪ element  {Add element to the set }
              element ⇐ next(flow, (S_l, S_j))
        END WHILE
        IF empty(Set_S_l) THEN return(Fail)     {Fail if set is empty }
        ELSE return(Success_and_Set_S_l );
```

Figure 7. Example of all

to be found. After locating S_1 in the OptTyp table, a set is created to hold all of the statements that meet the requirements. The function first_dep locates a statement that is flow dependent on S_j and marks the dependences so that future flow_dep calls do not re-examine dependences. Next, a WHILE loop is used to verify that the first element and any others found are not S_i. If the element meets all conditions, then it is added to the set. Once all verified elements are collected in Set_S_1, an IF statement determines the cardinality of the set. An empty set for an all quantifier indicates that failure should be returned to the driver.

Different combinations of the three quantifiers (any, no, and all) and the mem operator can be used in the specifications and result in the corresponding code segment being automatically generated.

EXPERIENCE WITH GENESIS

Four kinds of experiments performed using Genesis are reported here. First, the correctness of the generated code was examined by comparing optimizers produced by Genesis to hand-coded optimizers. Secondly, the efficiency of the generated code was examined by implementing and verifying a cost analysis of the optimizers. Thirdly, a number of experiments were performed to determine ways to improve the efficiency of optimizers. Fourthly, the frequency that several optimizations occur in practice was investigated.

A representative sample of eleven optimizations were chosen for experimentation: constant propagation (CTP), dead code elimination (DCE), constant folding (CFO), invariant code motion (ICM), loop unrolling (LUR), copy propagation (CPP), loop circulation (CRC), bumping (BMP), parallelization (PAR), loop fusion (FUS), and loop interchanging (INX). These eleven optimizations were applied to ten test programs to determine if the optimizations were performed correctly, if all application sites were found, and the frequency of their occurrence. Test programs were obtained from a HOMPACK test suite* and a numerical analysis test suite.[10] Ten programs were chosen that displayed programming constructs that might enable parallelization (i.e. loops that used arrays). No attempt was made to use programs in which certain optimizations might be applicable.

Experimental comparison of Genesis

The eleven optimizations were applied to the test programs and the optimized code was compared to the optimized code produced by a hand-generated optimizer, Tiny.[11] A comparison of Tiny's optimizations and the Genesis optimizations revealed that the automatically generated optimizations did not introduce any extraneous code into the optimized intermediate code. These first experiments successfully tested the Genesis system for correctness and produced evidence that the quality of code generated is equivalent to that of a hand-generated optimizer. Also, the robustness of Genesis is demonstrated by the ability to specify a wide range of optimizations.

* HOMPACK was obtained via anonymous ftp from netlib@mcs.anl.gov

Cost analysis of generated optimizers

In order to estimate the cost of applying optimizations, an analysis can be performed using the specification in Gospel. The cost analysis approximates the computational complexity of an optimization by assigning cost units to the primitive actions and dependency conditions that are involved in applying an optimization. By using the cost analysis, the user can approximate the cost without an actual implementation. Also, reduction in the cost of applying an optimization may be found and reduced by changing the specification. A technique to determine the cost of applying an optimization may be used to establish the cost of a proposed optimization from its specification. Each time a specification variable is referenced, a unit charge of one is assessed, as the lookup in the OptTyp table would be needed in the implementation of an optimizer. Likewise, each reference to a data dependence is assessed a unit charge. The primitive actions performed by the optimizer (i.e. move, copy, delete, add, modify) are assessed charges according to the relative cost of the operation.

In order to validate this approach, the cost analyses for seven optimizations were compared with their actual execution timings. These values are reported in Table III for CTP, DCE, CFO, LUR, CPP, FUS, and INX[12] (two implementations of FUS and INX: FUX_{SCA}, FUS_{VCA}, INX_{SCA}, and INX_{VCA} are reported and are described later). The Cost column indicates the cost assigned by the cost analysis and the Timing column is the actual execution timing (in microseconds) of applying the optimization.

A graph of the cost versus the execution timings of these optimizations is displayed in Figure 8. This graph reveals that the cost analysis is approximately linearly related to the machine timings of applying the optimization (the coefficient of regression for nine sample points is 0·968).

These execution timings suggest that the cost assignments are a realistic measure of the performance of optimizations. Also, the execution timings produced by a prototype implementation of Genesis demonstrate that the automatic generation of optimizers is a reasonable approach in the design and implementation of optimizers.

Table III. Cost versus execution timings

Optimization	Cost	Timing
CTP	11	1·0
DCE	12	1·8
CFO	27	2·6
LUR	23	3·8
INX_{VCA}	54	17·2
INX_{SCA}	181	23·6
FUS_{VCA}	329	44·2
CPP	434	65·4
FUS_{SCA}	485	81·4

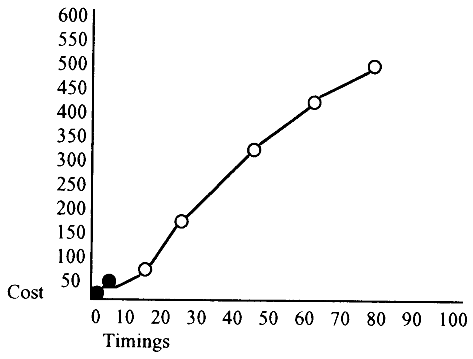

Figure 8. Graph of cost and execution timings

```
1. FOR each $S_m$ a member of the inner loop DO
    2. FOR each $S_n$ a member of the inner loop DO
        3. IF $S_n$ is flow dependent on $S_m$ with direction vector (<,>) THEN FAIL
4. FOUND
```

Figure 9. Statement comparison algorithm (SCA)

Alternate optimizer implementation

Through experimentation it was discovered that the generator implemented some optimizations such that unnecessary costs were incurred by the optimizer. Unnecessary overhead can occur when comparing the dependences and direction vectors of statements in various loops. There are two approaches to this comparison: (1) find all statements in the loops and then examine their dependences and direction vectors, or (2) for each statement in one loop, determine those dependences that terminate (i.e. the sink of the dependence arc) in the other loop and then examine those direction vectors.

The implementation described in this paper generated code for dependence testing that compared one statement involved in the test to all other statements in the range (i.e. the first approach). For example, the implementation of loop interchange is given in Figure 9.

Hence, the described implementation examined the dependences of all statements in the nested loops and then compared their direction vectors. An alternative implementation would examine the direction vectors of all statements that are flow dependent on S_m and if they were (<,>) would check to see if the statement was an element of the inner loop. The alternative implementation of loop interchanging is given in Figure 10.

Specifications that involve testing the membership of two or more specification

```
1. FOR each $S_m$ a member of the inner loop DO
    2. FOR each $S_n$ flow dependent on $S_m$
        3. IF the flow dependence direction vector is (<,>)
           and $S_n$ a member of the inner loop THEN FAIL
4. FOUND
```

Figure 10. Direction vector comparison algorithm (VCA)

variables could be implemented using either of the approaches depicted in these algorithms (e.g. FUS and INX). The difference in the performance of the two algorithms depends on the number of statements in the loop versus the number of dependences per statement that exist in the source program. Thus, when the optimization is being generated, it is not known which algorithm performs better.

A comparison of the performance of the two Algorithms is illustrated in Table IV by examining several applications of FUS and INX. The cost of using the statement comparison algorithm (SCA) is given in the first column for 4 applications of FUS and 4 applications of INX. Column 2 depicts the cost of using the Direction Vector Comparison Algorithm (VCA) at the same application sites. The last column is the percentage of reduction that VCA produces. In some cases, this percentage is negative indicating that VCA increases the optimization time.

In these experiments, when VCA performed better, it did so tremendously. On the average, the results with VCA were better, and in the case of FUS all applications using VCA were better. However, three out of four were worse for INX. Thus, there is not a clear indication that one algorithm is better than the other.

A heuristic that decides which implementation to choose could be incorporated as part of the optimizer. This heuristic needs several data items:

1. Number of data dependences per statement—d_i (known value, produced by data dependence analyzer).
2. Number of statements that occur in the loop—L (known value, produced by parser).
3. Estimate of the percentage of dependences that occur within loops—D (estimated by running numerous programs).

With this information, a heuristic was developed for the optimizer to select the implementation that performs better. The heuristic sums the number of data dependences that occur per statement in the loop and multiplies this number with the estimated percentage of dependences that emanate and terminate within loops (i.e. loop-carried dependences) to produce the number of data dependences that occur in a loop. This value is then compared to the number of statements within the loop as follows:

Table IV. Comparison of implementation techniques

Cost SCA	Cost VCA	Percentage reduction
FUS		
270	53	80
509	86	83
3758	108	97
2282	398	83
INX		
10	19	–90
589	132	78
34	41	–21
25	28	–12

$$\left(\sum_{i=0}^{L} d_i\right) D < L$$

If the inequality is true, then VCA is more efficient. In other words, if the estimated number of dependences within the loop is less than the number of statements, then fewer iterations would be made through step 2 of VCA than step 2 of SCA.

An estimate of the value of D was produced by automatically calculating the number of dependences that occurred between loops. This calculation was performed on ten test programs. The average percentage of loop dependences (i.e. D in the heuristic) in these test cases is 59. The heuristic was applied to the four INX test runs and correctly chose the algorithm to use in each of these cases.

The direction vector comparison algorithm was developed to improve the efficiency of the optimizer. However, after examining various specifications, some patterns that occur in the specification suggest that this algorithm may also become expensive. Identifying possible ‘costly’ specification patterns for VCA aides in the decision of which algorithm should be used to produce an efficient optimizer. If such patterns are found, a tool such as Genesis could produce both implementations and call on the appropriate technique dynamically.

Application frequency

When implementing an optimizer, the implementation team must decide what optimizations to apply. Part of this decision is based on how frequently the optimization to be included is expected to be applied. As data involving the application frequency of optimizations is helpful in the design of an optimizer, eleven optimizations were applied to the ten source programs. The results are displayed in Table V. The first column displays the number of application points that were found for the particular optimization. The second column displays the number of programs in which these application points were found. Notice that invariant code motion was not found, as these optimizations were applied to an extended intermediate code

Table V. Application frequency

	Application frequency	Program occurrences
DCE	34	5
CTP	97	6
CPP	5	2
CFO	5	1
ICM	0	0
LUR	49	4
FUS	11	4
INX	13	3
BMP	52	4
CRC	4	1
PAR	26	3

format where array references were not expanded. The large numbers of application points for DCE, CTP, and LUR are attributed to the enabling of these optimizations.[13]

These data suggest that the application of DCE, CTP, and LUR should be highly considered for application in an optimizer. It also suggests that ICM should not be applied when array references are explicit, but may be considered when array references are fully calculated. Also, the application of INX should be considered before CRC as it does not appear as often. Data may also be collected to determine how the optimizations interact, study the relationship between interactions and the ordering of the optimizations, and the effect these orderings have on the resultant code.[14]

CONCLUSION

This paper describes a prototype implementation of an automatic optimizer generator that produces optimizers from specifications. The Gospel specifications are used as input to the Genesis tool. The generator produces optimization specific code that is used in the construction of an automatically generated optimizer. A prototype of this tool was developed, verified, and used for experimentation. The automatic generation of an optimizer may be used for several purposes: the interaction of optimizations could be studied for determining possible orderings,[14] the cost and benefit of an optimization could be determined to decide if an optimization should be included in a production optimizer, and optimizations can be easily tailored using Genesis and, thus, a comparison of the effectiveness of the optimizations may be performed.

Experimentation revealed that the execution of optimizers generated by Genesis did not introduce any new code into the optimized intermediate code. Further experimentation suggests that although sequential optimizations may be applied to the entire program, parallelizing optimizations should be applied on a point-by-point basis since different orderings are needed in disjoint parts of the same program. Also, experimentation determined that the theoretical interactions[11] that may occur among optimizations do occur in practice.

acknowledgement

This work was partially supported by the U.S. National Science Foundation under Grant CCR-9109089 to the University of Pittsburgh.

APPENDIX I: OPTIMIZATIONS SPECIFIED IN GOSPEL

BMP:	Bumping—increases the loop bounds and all uses of the loop control variable by a set amount, the bump.
CFO:	Constant folding—replaces a binary operation with the result of that operation.
CPP:	Copy propagation—propagates copies to the uses of that variable.
COL:	Loop collapsing—changes a doubly nested loop into a singly nested loop.
CRC:	Loop circulation—interchanges the outermost and innermost loops of multiply nested loops (an extension of loop Interchanging).
CSE:	Common subexpression elimination—replaces a recomputation of a subexpression by assigning the subexpression to a variable that is used

	in its place.
CTP:	Constant propagation—propagates a constant to a single use.
DCE:	Dead code elimination—deletes a statement that defines a variable that is never used.
DIS:	Loop distribution—splits one loop into two adjacent loops (inverse of FUS).
FUS:	Loop fusion—converts two adjacent loops into one loop.
ICM:	Invariant code motion—moves loop invariant code outside of the loop.
INX:	Loop interchanging—interchanges two tightly nested loops.
IVE:	Inducation variable elimination—removes unnecessary induction variables and replaces them with computations involving one loop induction variable.
LUR:	Loop unrolling—unrolls a loop, forming two loops.
NRM:	Loop normalization—normalizes the step increment of a loop to be one.
PAR:	Parallelization—modifies DO to PAR, signifying parallel execution of the loop.
PEL:	Loop peeling—removes a single iteration from a loop.
SCE:	Scalar expansion—expands scalars to arrays when used with arrays in binary operations.
SMI:	Strip mining—replaces a single loop with two loops where the upper bound of the outer loop is the length of the vector register.
SRE:	Strength reduction—changes statements involving multiplication to addition.
UNS:	Loop unswitching—replaces a loop with a conditional by a conditional with two loops.

APPENDIX II. PRECOND GRAMMAR FOR THE GOSPEL PROTOTYPE

```
Precon          →   DEPEND Precon_list
Precon_list     →   Quantifier Code_list: Mem_list Condition_list; Precon_list | ε
Quantifier      →   ANY | NO | ALL
Code_list       →   StmtId StmtId_list
Mem_list        →   Mem_list OR Mem_list
                →   Mem_list AND Mem_list
                →   Mem(StmtId, Set_Exp)
Mem             →   MEM | NO_MEM
Set_Exp         →   INTER (Set_Exp, Set_Exp)
                →   UNION (Set_Exp, Set_Exp)
                →   ID
                →   PATH (ID, ID)
                →   CTRL_DEP (ID)
Condition_list  →   NOT Condition_list
                →   Condition_list AND Condition_list
                →   Condition_list OR Condition_list
                →   Type (StmtId, StmtId Dir_Vect)
                →   (StmtId Rel_Op StmtId)
Type            →   FLOW_DEP | OUT_DEP | ANTI_DEP | CTRL_DEP
Dir_Vect        →   (Dir Dir_List) | ε
```

```
Dir_List        →    , Dir | ε
Dir             →    Rel_Op | ANY
Rel_Op          →    < | > | <= | >= | = | !=
StmtId          →    ID | POS(ID)
StmtId_list     →    , StmtId | ε
```

REFERENCES

1. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung and D. A. Schouten, ‘Parafrase-2: an environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors’, *Proc. of 1989 International Conference on Parallel Processing*, St. Charles, Illinois, August 1989, pp. 39–48.
2. Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley and Haspal Subhlock, ‘The ParaScope editor: an interactive parallel programming tool’, *Proc. Supercomputing ’89*, Reno, Nevada, pp. 540–549.
3. F. E. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hseh and V. Sarkar, ‘A framework for determining useful parallelism’, *Proc. 1988 International Conference on Supercomputing*, St. Malo, France, February 1988, pp. 207–215.
4. Jack W. Davidson and Christopher W. Fraser, ‘Automatic generation of peephole optimizations’, *Proc. ACM SIGPLAN ’84 Symposium on Compiler Construction*, 1984, pp. 111–115.
5. Christopher W. Fraser and Alan L. Wendt, ‘Automatic generation of fast optimizing code generators’, *Proc. SIGPLAN ’88 Conference on Programming Language Design and Implementation*, June 1988, pp. 79–84.
6. Robert Giegerich, ‘Automatic generation of machine specific code optimizer’, *Proceedings of Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 75–81.
7. Robert R. Kessler, ‘Peep—an architectural description driven peephole optimizer’, *Proc. ACM SIGPLAN ’84 Symposium of Compiler Construction*, *SIGPLAN Notices*, **19**, (6), 106–110 (1984).
8. Deborah Whitfield and Mary Lou Soffa, ‘Automatic generation of global optimizations’, *ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, June, 1991, pp. 120–129.
9. Tia Watts, Mary Lou Soffa and Rajiv Gupta, ‘Techniques for integrating parallelizing transformations and compiler based scheduling methods’, *Supercomputing ’92*, Minneapolis, MN, 1992.
10. Richard Burden and J. Douglas Faires, in *Numerical Analysis*, Prindle, Weber & Schmidt, Boston, MA, 1989.
11. Michael Wolfe, ‘Tiny: a loop restructuring research tool’, Oregon Graduate Institute of Science and Technology, 1989.
12. David A. Padua and Michael J. Wolfe, ‘Advanced compiler optimizations for supercomputers’, *Communications of the ACM*, **29**, (12), 1184–1201 (1986).
13. Deborah Whitfield and Mary Lou Soffa, ‘An approach to ordering optimizing transformations’, *Proc. Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, March 1990, pp. 137–146.
14. Deborah Whitfield and Mary Lou Soffa, ‘Investigation properties of code transformations’, *Proc. 1993 International Conference on Parallel Processing*, St. Charles, Illinois, August, 1993, Vol. 2, pp. 156–160.