An Approach for Exploring Code Improving Transformations*

Deborah L. Whitfield** and Mary Lou Soffa
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT:

Although code transformations are routinely applied to improve the performance of programs for both scalar and parallel machines, the properties of code improving transformations are not well understood. In this paper we present a framework that enables the exploration, both analytically and experimentally, of properties of code improving transformations. The major component of the framework is a specification language, Gospel, for expressing the conditions needed to safely apply a transformation and the actions required to change the code to implement the transformation. The framework includes a technique that facilitates an analytical investigation of code improving transformations using the Gospel specifications. It also contains a tool, Genesis, that automatically produces a transformer that implements the transformations specified in Gospel. We demonstrate the usefulness of the framework by exploring the enabling and disabling properties of transformations. We first present analytical results on the enabling and disabling properties of a set of code transformations, including both traditional and parallelizing transformations, and then describe experimental results showing the types of transformations and the enabling and disabling interactions actually found in a set of programs.

1. Introduction

Although code improving transformations have been applied by compilers for many years, the properties of these transformations are not well understood. It is widely recognized that the place in the program code where a transformation is applied, the order of applying code transformations, and the selection of the particular code transformation to apply can have an impact on the quality of code produced. Although concentrated research efforts have been devoted to the development of particular code improving transformations, the properties of the transformations have not been adequately identified or studied. This is due in part to the informal methods used to describe code improving transformations. Because of the lack of common formal language or notation, it is difficult to identify properties of code transformations, to compare transformations and to determine how transformations interact with one another.

By identifying various properties of code improving transformations, such as their interactions, costs, expected benefits, and application frequencies, informed decisions can be made as to what transformation to apply, where to apply them, and in which order to apply them. The order of application is important to the quality of code as transformations can interact with one another by creating or destroying the potential for further code improving transformations. For

* This work was partially supported by NSF under grant CCR-9407061 to Slippery Rock University and by CCR-9109089 to the University of Pittsburgh.

** Dr. Whitfield's address is Department of Computer Science, Slippery Rock University, Slippery Rock, PA 16057

example, the quality of code produced would be negatively affected if the potential for applying a beneficial transformation was destroyed by the application of a less beneficial transformation. Certain types of transformations may be beneficial for one architecture but not for another. The benefits of a transformation can also be dependent on the type of scheduler (dynamic or static) that is used.[15]

One approach that can be taken to determine the most appropriate transformations and the order of application for a set of programs is to implement a code transformer program (optimizer) that includes a number of code improving transformations, apply the transformations to the programs, and then evaluate the performance of the transformed code. However, actually implementing such a code transforming tool can be a time consuming process, especially when the detection of complex conditions and global control and data dependency information is required. Also, because of the ad hoc manner in which such code transformers are usually developed, the addition of other transformations or even the deletion of transformations may necessitate a substantial effort to change the transformer. Another approach is to modify an existing optimizer. However, optimizing compilers are often quite large (e.g., SUIF[12] is about 300,000 lines of C++ code and the GNU C compiler[8] is over 200,000 lines of code) and complex, making it difficult to use them in experiments that take into account the various factors influencing the performance of the transformed code.

In this paper, we present a framework for exploring properties of code improving transformations. The major component of the framework is a code transformation specification language, Gospel. The framework includes a technique that utilizes the specifications to analytically investigate the properties of transformations. Gospel is also used in the design of Genesis, a tool that automatically produces a code transformer program from the specifications, enabling experimentation. A specification for a transformation consists of expressing the conditions in the program code that must exist before the transformation can be safely applied and the actions needed to actually implement the transformation in the program code. The specification uses a variant of first order logic and includes the expression of code patterns and global data and control dependencies required before applying the transformation. The actions are expressed using primitive operations that modify the code. The code improving transformations that can be expressed in Gospel are those that do not require a fix-point computation. This class includes many of the traditional and parallelizing code improving transformations.

We demonstrate how the framework can be used to study the phase ordering problem of transformations by exploring the enabling and disabling properties of transformations. Using Gospel, we first show that enabling and disabling properties can be established analytically. We also demonstrate through the use of Genesis that these properties can be studied experimentally. Using Genesis, code transformers were automatically produced for a set of transformations specified in Gospel, and then executed to transform a test suite of programs. We present results on experiments that explored the kinds of transformations found in the test suite and the types and numbers of transformation interactions that were found.

A number of benefits accrue from such a framework. Guidelines suggesting an application order for a set of code improving transformations can be derived from both the analytical and experimental exploration of the interactions. Also, a new transformation can be specified in Gospel and its relationship to other transformations analytically and experimentally investigated. From the specifications, a transformer can be generated by Genesis and using sample source programs, the user can experimentally investigate transformations on the system under consideration. The decision as to which transformations to include for a particular architecture and the order in which these transformations should be applied can be easily explored. New transformations that are particularly tailored to an architecture can be specified and used to generate a transformer. The effectiveness of the transformations can be experimentally determined using the architecture. Transformations that are not effective can be removed from consideration, and a new transformation can be added by simply changing the specifications and rerunning Genesis, producing a program (transformer) that implements the new transformation. Transformations that can safely be combined could also be investigated analytically and the need to combine them can be explored experimentally. Another use of Gospel and Genesis is as a teaching tool. Students can write specifications of existing transformations, their own transformations, or can modify and tune transformations. Implementations of these transformations can be generated by Genesis, enabling experimentation with the transformations.

Prior research has been reported on tools that assist in the implementation of code improving transformations, including the analysis needed. Research has been performed on automatic code generation useful in the development of peephole transformers.[4,6,7,9] In these works, the transformations considered are localized and require no global data flow information. A number of tools have been designed that can generate analyses. Sharlit[13] and PAG[1] use lattice-based specifications to generate global data-flow analyses. SPARE is another tool that facilitates the development of program analysis algorithms[14]. This tool supports a high level specification language through which analysis algorithms are expressed. The denotational nature of the specifications enables automatic implementation as well as verification of the algorithms. A software architecture useful for the rapid prototyping of data flow analyzers has also recently been presented.[5]

Only a few approaches have been developed that integrate analysis and code transformations, which our approach does. A technique to combine specific transformations by creating a transformation template that fully describes the combined operations was developed as part of the framework for iteration-reordering loop transformations.[11] New transformations may be added to the framework by specifying new rules. This work is applied only to iteration-reordering execution order of loop interactions in a perfect (tight) loop nest and does not provide a technique to specify or characterize transformations in general.

The next section of this paper discusses the framework developed to specify transformations. Section 3 presents details of the Gospel language. Section 4 shows how Gospel can be used in the analytical investigation of the enabling and disabling conditions of transformations, and in the automatic generation of transformers. Section 5 demonstrates the utility of the specification technique using Genesis and presents experimental results. Conclusions are presented in Section 6.

2. Overview of the Transformation Framework

The code improving transformation framework, shown in Figure 1, has three components: Gospel, a code transformation specification language; an analytical technique that uses Gospel specifications to facilitate formal proofs of transformation properties; and Genesis, a tool that uses the Gospel specifications to produce a program that implements the application of transformations. These three components are used to explore transformations and their properties. In this paper, we use the framework to explore disabling and enabling properties.

A Gospel specification consists of the preconditions needed in program code in order for a transformation to be applicable, and the code modifications that implement the transformation. Part of the precondition specification is the textual code pattern needed for a transformation. An example includes the existence of a statement that assigns a variable to a constant or the existence of a nested loop. Thus, the code patterns operate on program objects, such as loops, statements, expressions, operators and operands.

In order to determine whether it is safe to apply a transformation, certain data and control dependencies may also be needed. Program objects are also used to express these dependence relationships. In describing transformations, Gospel uses dependencies expressed in terms of flow, anti, output, and control dependencies.[21] These dependencies are quantified and combined using logical operators to produce complex data and control conditions. A flow dependence ($S_i\ \delta\ S_j$) is a dependence between a statement that defines a variable and a statement S_j that uses the definition in S_i. An anti-dependence ($S_i\ \delta^-\ S_j$) exists between statement S_i that uses a variable that is then defined in statement S_j. An output dependence ($S_i\ \delta^o\ S_j$) defines a dependence between a statement S_i that defines (or writes) a variable that is later defined (or written) by S_j. A control dependence ($S_i\ \delta^c\ S_j$) exists between a control statement S_i and all of the statements S_j under its control. The concept of data direction vectors for both forward and backward loop-carried dependencies of array elements is also needed in transformations for parallelization.[10] Each element of the data dependence vector consists of either a forward, backward, or equivalent direction represented by <, >, or =, respectively. These directions can be combined into >=, <=, and *, with * meaning any direction. The number of elements in the direction vector corresponds to the loop nesting level of the statements involved in the dependence.

In some cases, code improving transformations have been traditionally expressed using global data flow information. This information can either be expressed as a combination of the data and control dependencies[21] or can be introduced in Gospel as a relationship that needs to be computed and checked. The underlying assumption of Gospel is that any algorithm needed to compute the data flow or data dependency information is available. Thus, Gospel uses basic control and data dependency information with the possibility of extensions to other types of data flow information. It should be noted that in the more than twenty transformations studied in this research, all data flow information was expressed in terms of combinations of data and control dependencies.[16, 17] A sample of transformation specifications is given in Appendix B.

Gospel also includes the specification of the code modifications needed to implement a transformation. Although code improving transformations can produce complex code modifications, the code changes are expressed in Gospel by primitive operations that can be applied in combinations to specify complex actions. These operations are applied to code objects such as statements, expressions, operands and operations. Using primitive operations to express code modifications provides the flexibility to specify a wide range of code modifications easily.

Another component of the framework is an analytical technique useful for proving properties of transformations. The technique uses the specification from Gospel to provide a clear, concise description of a transformation useful in analysis. We show how this component was used in establishing the enabling and disabling properties of a set of transformations.

The last component of the framework is Genesis, a tool that generates a program that implements transformations from the Gospel specification of those transformations. Thus, the generated program contains code that will check that conditions needed for the safe application of a transformation are satisfied and also contains code that will perform the code modifications as expressed in the Gospel specification. A program to be transformed is then input into the program generated by Genesis and the output produced is the program transformed by the specified transformations. A run-time interface is provided that either permits the user to select the type and place of application of a transformation, or it automatically finds all applicable transformations at all points. We demonstrate the utility of Genesis in determining the kinds and frequencies of transformations occurring in a number of programs, and the types and frequencies of enabling and disabling interactions.

Figure 1 presents the code improving framework and uses of the framework. The three components of the framework are shown in the box and some applications of the framework are shown in ovals. Solid lines connect the framework with the applications that are described in this paper. A solid line connects the framework to the interaction prover used to establish enabling and disabling properties of transformations. There is another solid line between the framework and the experimental studies of enabling and disabling properties. The dotted line connecting the framework and the combining transformations represents a potential use of the framework yet to be fully explored.

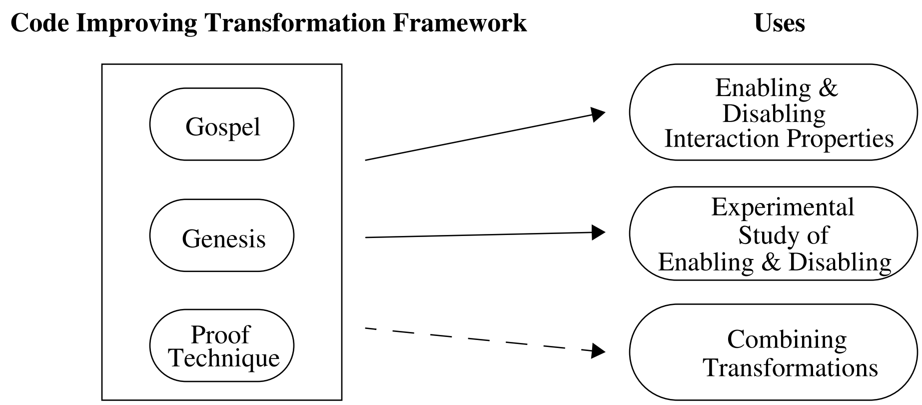

Figure 1. Components and Utilization of the Transformation Framework

3. Description of the Gospel Language

Gospel is a declarative specification language capable of specifying a class of transformations that can be performed without using fix-point computation. We have specified over twenty transformations using Gospel, including specifications for invariant code motion, loop fusion, induction variable elimination, constant propagation, copy propagation and loop unrolling. Transformations that do require fix-point computation such as partial dead code elimination and partial redundancy elimination cannot be specified. Likewise, although Gospel can be used to specify a type of constant propagation and folding, it cannot be used, for example, to specify constant propagation transformations requiring fixed point computation. However, studies have shown that code seldom contains the types of optimizations needing iteration.[3] A BNF grammar for a section of Gospel appears in Appendix A. The grammar is used to construct well-formed specifications and also used in the implementation of the Genesis transformer.

In this paper, we assume the general form of statements in a program to be transformed is three address code extended to include loop headers and array references. However, Gospel and Genesis can be adapted to handle other representations including source level representation. We assume that a basic three address code statement has the form:

$$\textbf{operand}_1 := \textbf{operand}_2\ \textbf{opcode}\ \textbf{operand}_3$$

The three address code retains the loop headers and array references from the source program, which enables the user to specify loop level transformations and array transformations.

The template for a specification of a transformation consists of a *Name* that is used to identify the particular code improving transformation followed by three major specification sections identified by keywords: DECLARATION, PRECONDITION and ACTION. The PRECONDITION section is decomposed into two sections, Code_Pattern and Depend. The overall design of a Gospel specification follows.

DECLARATION
PRECONDITION
    Code_Pattern
    Depend
ACTION

The DECLARATION section is used to declare variables whose values are code objects of interest (e.g., loop, statement). Code objects have attributes as appropriate such as a head for a loop and position for an operand. The PRECONDITION section contains a description of the code pattern and data and control dependence conditions, and the ACTION section consists of combinations of primitive operations to perform the transformation.

Figure 2 presents a Gospel specification of a Constant Propagation (CTP) transformation (See Section 3.2 for details). The specification uses three variables S_i, S_j and S_l whose values are statements. The Code_Pattern section specifies the code pattern consisting of **any** statement **S_i.opcode == assign** that defines a constant **type (S_i.opr$_2$) == const**. S_i will have as its value such a statement if it exists. In the Depend section, S_j is used to determine which statement uses the constant. The **pos** attribute records the operand position (first, second or third) of the flow dependence between S_i and S_j. The second statement with S_l ensures that there are **no** other definitions of the constant assignment that might reach S_j. Again, the **pos** attribute records the position of the flow dependence between S_j and S_l. The **S_j != S_l** specification indicates that the two statements are not the same statement and the **operand (S_j, pos) != operand (S_l, pos)** specification ensures that the dependence position recorded in S_j does not involve the same variables as the dependence found in S_1.

DECLARATION
    Statement: S_i, S_j, S_l;
PRECONDITION
  Code_Pattern   *Find a constant definition*
    any S_i: S_i.opcode == assign
    AND type (S_i.opr$_2$) == const;
  Depend   *Use of S_i with no other definitions*
    any (S_j, pos): flow_dep (S_i, S_j, (=));
    no (S_l, pos): flow_dep (S_l, S_j, (=)) AND (S_i != S_l)
    AND operand (S_i, pos) != operand (S_l, pos);
ACTION   *Change use in S_i to be constant*
    modify (operand (S_j, pos), S_i.opr$_2$);

Figure 2. Gospel Specification of Constant Propagation

If a S_j is found that meets the requirements and no S_l's are found that meet the specified requirements, then the operation expressed in the ACTION section is performed. The action is to modify the use at S_j to be the constant found as the second operand of S_i.

Next consider the specification of the parallelizing transformation Loop Circulation (CRC) found in Figure 3 that defines two statements and three tightly (perfect) nested loops, which are loops without any statements occurring between the headers. In the Code_Pattern section, **any** specifies an occurrence of tightly nested loops L_1, L_2, and L_3. The data dependence conditions in the Depend section first ensure that the loops are tightly nested by specifying no flow dependences between loop headers. Next, 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 loops are interchanged as specified in the ACTION section.

The next section provides more details about the Gospel language.

```
DECLARATION
        Statement: Sn, Sm;
        Tight: (L1, L2, L3);
PRECONDITION
  Code_Pattern                            Find Tightly nested loops
        any (L1, L2, L3);
  Depend                                  Ensure perfect nesting, no flow_dep with<,>
        no L1.Head: flow_dep (L1.Head, L2.Head);
        no L2.Head: flow_dep (L2.Head, L3.Head);
        no Sm, Sn: mem (Sm, L3) AND mem (Sn, L3),
        flow_dep (Sn, Sm, (<,>));
ACTION                                    Interchange the loops
        move (L1.Head, L3.Head);
```

Figure 3. Gospel Specification of Loop Circulation

3.1. Gospel Types and Operations

Variables, whose values are code elements, are defined in the declaration section and have the form

DECLARATION: *id_list*.

Variables are declared to be one of the following types: **Statement**, **Loop**, **Nested** loops, **Tight** loops, or **Adjacent** loops. Thus, objects of these types have as their value a pointer to a statement, loop, nested loop, tight loop or adjacent loop, respectively. All types have pre-defined attributes denoting relevant properties, such as next (**nxt**) or previous (**prev**). The usual numeric constants (integer and real) are available in Gospel specifications. Besides these constants, two classifications of pre-defined constants are also available: operand types and opcode values. These constants reflect the constant values of the code elements that are specified in Gospel. Examples of constants include **const** for a constant operand and **var** for a variable operand. Typical mathematical opcodes as well as branches and labels can appear in the specification code. Gospel can be extended to include other op codes and variable types by changing the grammar and any tools, such as Genesis, that uses the grammar.

A variable of type **Statement** can have as its value any of the statements in the program and possesses attributes indicating the first, second and third operand (**opr**$_\mathbf{1}$, **opr**$_\mathbf{2}$, and **opr**$_\mathbf{3}$, respectively) and the operation (**opcode**). Additionally a **pos** attribute exists to maintain the operand position of a dependence required in the Depend section. A **Loop** typed variable points to the header of the loop, and has as attributes **Body**, which identifies all the statements in the loop and **Head,** which defines **Lcv,** the loop control variable, **Init**, the initial value and **Final**, the last value of the loop control variable. The **End** of the loop is also an attribute. Thus, a typical loop structure, with its attributes is:

Head	{*L.Head defines L.Init, L.Final, and L.Lcv*}
Start_of_Loop	{*L.Start*}
Loop_body	{*L.Body*}
End_of_Loop	{*L.End*}

Nested loops, **Tight** loops, and **Adjacent** loops are composite objects whose components are of type **Loop**. **Nested** loops are defined as two (or more) loops where the second named loop appears lexically within the first named loop. **Tight** loops restrict nested loops by ensuring that there are no statements between loop headers. **Adjacent** loops are nested loops without statements between the end of one loop and the header of the next loop.

The *id_list* after the keyword DECLARATION is either a simple list (e.g., statement and loop identifiers) or a list of pairs (e.g., identifiers for a pair of nested, adjacent or tight loops). For example, **Tight: (Loop_One, Loop_Two)** defines a loop structure consisting of two tightly nested loops.

3.2. The Gospel Precondition Section

In order to specify a code improving transformation and conditions under which it can be safely applied, the pattern of code and the data and control dependence conditions that are needed must be expressed. These two components constitute the precondition section of a specification. The keyword PRECONDITION is followed by the keywords Code_Pattern, which identifies the code pattern specifications, and Depend which identifies the dependence specification.

Code Pattern Specification

The code pattern section specifies the format of the statements and loops involved in the transformation. 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 can be one of **any**, **all** or **no** with the following meanings:

all - returns a set of all the elements of the requested types for a successful match

any - returns a set of one element of the requested type if a match is successful

no - returns a null set if the requested match is successful

For example, the quantifier element list **any (S_j)** returns a pointer to some statement S_j.

The second part of the code pattern specification *format_of_elements* describes the format of the elements required. If **Statement** is the element type, then *format-of-elements* restricts the statement's operands and operator. Similarly, if **Loop** is the element type, *format-of-elements* restricts the loop attributes. 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*. An example code pattern specification which specifies that the final iteration count is greater than the initial value is:

any Loop: Loop.Final - Loop.Init > 0

Expressions can be constructed in *format_of_elements* using the **and** and **or** operators with their usual meaning. Also, restrictions can be placed on either the **type** of an operand (i.e., **const**, or **var**) or the position, **pos**, of the opcode as seen in the Code_Pattern section of Figure 2.

Depend Specification

The second component of the PRECONDITION section is the Depend section, which specifies the required data or control dependencies of the transformation. The dependence specification consists of expressions quantified by **any**, **no**, or **all** that return both a Boolean truth value and the set of elements that meet the conditions. If the **pos** attribute is used, then the operand position of the dependence is also returned. The general form of the dependence specification is:

quantifier element: *sets_of_elements*, *dependence_conditions*

The *sets_of_elements* component permits specifying set membership of elements; **mem**(*Element*, *Set*) specifies that *Element* is a member of the defined *Set*. *Set* can be described using predefined sets, the name of a specific set, or an expression involving set operations and set functions such as union and intersection. The *dependence_conditions* clause describes the data and control dependencies of the code elements and takes the form:

type_of_dependence (StmtId, StmtId, Direction).

In this version of Gospel, 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; also <= and >=, can be used), or **any**, which allows any direction. Direction vectors are needed to specify loop-carried dependencies of array elements for parallelizing transformations. This direction vector may be omitted if loop-carried dependencies are not relevant.

As an example, the following specification is for one element named S_i that is an element of $Loop_1$ such that there is a S_j, an element of $Loop_2$, and there is either a flow dependence or an anti-dependence between S_i and S_j.

any S_i: mem(S_i, $Loop_1$) AND mem(S_j, $Loop_2$),

flow_dep(S_i, S_j,(=)) OR anti_dep(S_i, S_j, (=))

3.3. The Gospel Action Section

We decompose the code modification effects of applying transformations into a sequence of five primitive operations, the semantics of which are indicated in Table 1. These operations are overloaded in that they can apply to different types of code elements. The five primitive operations, their parameters and semantics are:

Operation	Parameter	Semantics
Move	(Object, After_Object)	*move* Object and place it following After_Object
Add	(Obj_Desc, Obj_Name, After_Obj)	*add* Obj_Name *with* Obj_Desc*, place it* after_Obj
Delete	(Object)	*delete* Object
Copy	(Obj, After_Obj, New_Name)	*copy* Obj into New_Name*, place* it After_Obj
Modify	(Object, Object_Description)	*modify* Object *with* Object_Desc

Table 1. Action Operations

An example of a Move operation that moves Loop_1 header after Loop_2 header is:

move(Loop_1.Head, Loop_2.Head).

An example of a modify action that modifies the end of Loop_2 to jump to the header of Loop_2 is:

modify(Loop_2.End, address (Loop_2.Head)).

These primitive operations are combined to fully describe the actions of a transformation. It may be necessary to repeat some actions for statements found in the PRECONDITION section. 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.

4. Applications of the Gospel Specification

The Gospel specifications are useful in a number of ways. In this section, we demonstrate the utilization of the specifications to explore the phase ordering problem of transformations by analytically establishing enabling and disabling properties. In Section 4.2, we show how Gospel is used to produce an automatic transformer generator, Genesis, which can be used to explore properties of transformations experimentally.

4.1. Technique to Analyze Specifications

The Gospel specifications can be analyzed to determine properties of transformations, and in particular, we use the analysis technique for establishing enabling and disabling properties of transformations. Through the enabling and disabling conditions, the interactions of transformations that can create conditions and those that can destroy conditions for applying other transformations are determined. Knowing the interactions that occur among transformations can be useful in determining when and where to apply transformations. For example, a strategy might be to apply a transformation that does not destroy conditions for applying another transformation in order to exploit the potential of the second transformation, especially if the second transformation is considered to be more beneficial.

4.1.1. Enabling and Disabling Conditions

Enabling interactions occur between two transformations when the application of one transformation creates the conditions for the application of another transformation that previously could not be applied. Disabling interactions occur when one transformation invalidates conditions that exist for applying another transformation. In other words, transformation A enables transformation B (denoted A $\Rightarrow$ B) if before A is performed, B is not applicable, but after A is performed, B can now be applied (B's pre-condition is now true). Similarly, transformation A disables transformation B (denoted A $\stackrel{D}{\Rightarrow}$ B) if the pre-conditions for both transformation A and B are true, but once A is applied, B's pre-condition becomes false. These properties are involved in the phase ordering problem of transformations.

Before determining the interactions among transformations, the conditions for enabling and disabling each transformation must be established. The enabling and disabling conditions are found by analyzing the PRECONDITION specifications of the transformations. For each condition in the Code_Pattern and Depend section of a transformation, at least one enabling/disabling condition is produced. For example, if a code pattern includes:

any Statement: Statement.opcode == assign

then the enabling condition is the creation of a statement with the opcode of assign, or the disabling conditions are the deletion of such a statement or the modification of the statement's opcode. The enabling and disabling conditions of six transformations derived from their specifications (see appendix B for their Gospel specifications) are given in Table 2.

4.1.2 Interactions Among Transformations

Using the Gospel specifications, we can prove the non-existence of interactions. We also use the specifications in developing examples that demonstrate the existence of interactions. Such an

Transformation	Enabling Conditions	Disabling Conditions
Dead Code Elimination (DCE)	1. Create S_i that is not used 2. Non-existence of S_l with (S_i δ= S_l) Delete S_l, Path is deleted*	1. Destroy S_i that is not used 2. Existence of S_l with (S_i δ= S_l) Introduce S_l that uses value computed by S_i
Constant Propagation (CTP)	1. Create S_i 2. Insert S_j such that (S_i δ= S_j) 3. Non-existence of S_l with (S_l δ= S_j) Modify S_l so that S_l == S_i, Destroy (S_l δ= S_j): a) Introduce a definition*, b) Delete S_l*, c) Path is deleted*	1. Destroy S_i 2. Non-existence of S_j with (S_i δ= S_j) Delete S_j, Move S_j* 3. Existence of S_l with (S_l δ= S_j) Modify S_l so that $S_l \neq S_i$, Create (S_l δ= S_j): a) Definition is deleted*, b) Introduce S_l, where $S_l \neq S_i$ c) Path from S_l to S_j is created
Constant Folding (CFO)	1.Create S_i of the form CONST opcode CONST	1. Remove or Modify S_i
Loop Unrolling (LUR)	1. Create DO Loop, L	1. Destroy DO loop, L
Loop Fusion (FUS)	1. Existence of 2 adjacent loops: Add a loop 2. Two DO loops have identical headers: Modify a header 3. The non-existence of S_n and S_m with a backward dependence before a forward: Remove S_n or S_m Add definition between S_n, S_m* Delete path between S_n and S_m 4. Non existence of (S_i δ= S_j) Remove S_j Remove S_i * Add def., destroying depend* Delete path between S_i and S_j*	1.Existence of 2 non-adjacent loops: Add a loop 2.Two DO loops do not have identical headers: Modify a header 3.The existence of S_n and S_m with a backward dependence before a forward: Insert S_n or S_m Delete definition between S_n, S_m* Create path between S_n and S_m 4. Existence of (S_i δ= S_j) Insert S_j Delete a def., so dependence holds* Create path between S_i and S_j
Loop Interchanging (INX)	1. Existence of 2 nested DO loops: Add a loop 2. Non-existence of S_n, S_m with a (<,>) dependence: Remove S_n* or S_m Add definition between S_n, S_m* Delete path between S_n and S_m 3. Loop headers are invariant: Modify a header	1.Non-Existence of 2 nested DO loops: Remove a loop 2. Existence of S_n, S_m with a (<,>) dependence: Insert S_n or S_m Remove def. between S_n, S_m* Create path between S_n and S_m 3.Loop headers vary with respect to each other: Modify a header

* denotes condition is not possible in correct specifications (i.e., maintains semantic equivalence)

Table 2. Enabling and Disabling Conditions

example of an interaction is given in Figure 4, where Loop Fusion (FUS) enables Loop Interchange (INX). The two inner loops on J are fused into one larger loop, which can then be interchanged.

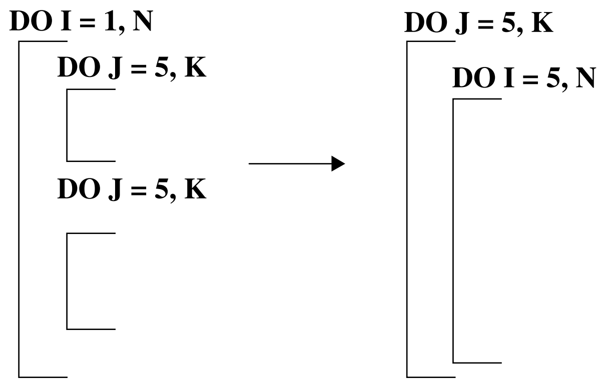

Figure 4. Loop Fusion Enables Loop Interchanging

Sometimes the interaction between two transformations is more complex in that a transformation can both enable and disable a transformation. Invariant Code Motion (ICM) and Loop Interchange (INX) are two such transformations, as shown in Figure 5. ICM enables INX and also can disable INX. In Figure 5 (a), an example of ICM enabling INX is given and in Figure 5 (b) an example of ICM disabling INX is shown.

For ease in proving the non-interaction, we use a formal notation of the Gospel specifications that is directly derived from the specification language by using mathematical symbols in place of the language related words. A comparison of the two styles is exemplified by:

Language: no S_m, S_n: mem (S_m, L_2) AND mem (S_n, L_2), flow_dep (S_n, S_m, (<,>));

Formal: $\neg\exists\, S_m, S_n$ such that $S_m \in L_2 \wedge S_n \in L_2 \wedge (S_n\ \delta_{<,>}\ S_m)$

```
for i = 1 to 100                        a= b * 2 + 4
 a = b * 2 + 4                          for i = 1 to 100
   for j = 1 to 100        ------>        for j = 1 to 100
   ...                                    ...
  end j                                   end j
end i                                   end i
```

(a) ICM enables INX

```
for i = 1 to 100                        for i = 1 to 100
   for j = 1 to 100                     a= i * 2 + 4
   a = i * 2 + 4           ------>        for j = 1 to 100
    ...                                   . ..
   end j                                  end j
end i                                   end i
```

(b) ICM disables INX

Figure 5: Enabling and Disabling Transformations

The following claim and proof illustrate the technique to prove non-existence of enabling and disabling interactions between transformations. The claim is that loop interchange (INX) cannot disable the application of constant propagation (CTP). The proof utilizes the disabling conditions for CTP as given previously in Table 2.

Claim: INX $\neg\overset{D}{\Rightarrow}$ CTP (Loop interchange does not disable constant propagation.)

Proof: Assume that INX $\overset{D}{\Rightarrow}$ CTP. For INX to disable CTP, both INX and CTP must be applicable before INX is applied.

For INX to be applicable, there must be two tightly nested loops, L_1 and L_2 where the loop limits are invariant and there is no data dependence with a (<,>) direction vector.

For CTP to be applicable, there must exist a S_i that defines a constant and S_j which uses the constant value such that $(S_i\ \delta_=\ S_j) \wedge \neg\exists\ S_l$ such that $S_l\ \delta_=\ S_j$.

Since CTP is applicable, INX must alter the state of the code to disable CTP. The three disabling conditions for CTP given in Table 2, produce the following cases:

Case 1: Destroy S_i which defines the constant

INX does not delete any statements, but does move a header, L_2. S_i defines a variable and a loop header only defines the loop control variable. If the loop control variable and the variable defined in S_i were the same, then CTP is not applicable because S_i does not define a constant value. $\therefore$ INX does not destroy S_i, the statement defining the constant.

Case 2: The non-existence of S_j or the removal of the dependence $(S_i\ \delta_=\ S_j)$

INX does not delete any statements but does move a header, L_2. However, moving the header to the outside of the loop would not destroy the relationship $(S_i\ \delta_=\ S_j)$ since the headers must be invariant relative to each other in order for INX to be applicable. $\therefore$ INX does not destroy S_j.

Case 3: The creation of S_l such that $(S_l\ \delta_=\ S_j)$

INX does not create or modify a statement. So there are three ways for INX to create the condition $(S_1\ \delta_=\ S_j)$:

1) INX could delete a definition S_i, but this is not a legal action for this transformation.

2) INX could introduce S_l. INX does not create any statements, but it does move a header. S_l could not be the header because S_l defines a constant.

3) INX creates a path so that S_l reaches S_j. S_j could be the header, but the definition in S_l would have reached S_j prior to INX since the headers must be invariant. $\therefore$ INX does not create S_l.

Thus, we show that INX $\neg\overset{D}{\Rightarrow}$ CTP.♦

That is, loop interchange when applied will not destroy any opportunities for constant propagation.

By exploring examples of interactions and developing proofs for non-interaction, we derived (by hand) an interaction table that displays the potential occurrence of interactions. Table 3 displays interactions for eight transformations: Dead Code Elimination (DCE), Constant Propagation(CTP), Copy Propagation (CPP), Constant Folding (CFO), Invariant Code Motion (ICM), Loop Unrolling (LUR), Loop Fusion (FUS), and Loop Interchange (INX). Each entry in the table consists of two elements separated by a slash (/). The first element indicates the enabling relationship between the transformation labeling the row and the transformation labeling the column, and the second element is the disabling relationship. A "-" indicates that the interaction does not occur, whereas an "E" or "D" indicates that an enabling or disabling interaction occurs, respectively. As an example, the first row indicates that DCE enables DCE and disables CTP. Notice the high degree of potential interactions among the triples <FUS, INX, and LUR>, and <CTP, CFO, and LUR>.

4.1.3 Impact of the Interactions on Transformation Ordering

	DCE	**CTP**	**CPP**	**CFO**	**ICM**	**LUR**	**FUS**	**INX**
DCE	E/-	-/D	E/D	-/-	E/-	-/-	E/-	E/-
CTP	E/-	-/-	-/D	E/-	E/-	E/-	E/-	E/-
CPP	-/-	E/-	-/-	-/-	E/D	-/-	E/-	-/-
CFO	-/-	E/-	-/-	-/-	-/-	E/-	E/-	-/-
ICM	-/-	-/-	-/-	-/-	E/-	-/-	E/D	E/D
LUR	E/-	E/-	E/-	E/-	-/-	E/-	E/D	E/D
FUS	-/-	-/D	-/-	-/-	-/D	-/D	E/D	E/D
INX	-/-	-/-	-/-	-/-	E/D	-/-	E/D	E/D

Table 3. Theoretical Enabling and Disabling Interactions

The disabling and enabling relationships between transformations can be used when transformations are applied automatically or when transformations are applied interactively. When transformations are applied automatically, as is the case for optimizing compilers, the interactions can be used to order the application so as to apply as many transformations as possible. When applying transformations in an interactive mode, knowledge about the interaction can help the user determine which transformation to apply first. Using the interaction properties, two rules are used for a particular ordering, if the goal is to applying as many transformations as possible.

1. If transformation A can *enable* transformation B, then order A *before* B -- <A,B>.
2. If transformation A can *disable* transformation B, then order A *after* B -- < B, A>.

These rules cannot produce a definite ordering as conflicts arise when:

1. A $\Rightarrow$ B and B $\Rightarrow$ A
2. A $\stackrel{D}{\Rightarrow}$ B and B $\stackrel{D}{\Rightarrow}$ A
3. A $\stackrel{D}{\Rightarrow}$ B and A $\Rightarrow$ B

In these cases, precise orderings cannot be determined from the properties. However, as shown in the next section, experimentation can be performed using Genesis to determine if there is any value in applying one transformation before the other transformation.

As an example of using the orderings, consider a scenario where the transformer designer decides that LUR is an extremely beneficial transformation for the target architecture. The transformer designer could benefit from two pieces of information: 1) the transformations that enable LUR, and 2) the transformations that disable LUR. As can be seen in Table 3, CTP, CFO, and LUR all enable LUR. These interactions indicate that CTP and CFO should be applied prior to LUR for this architecture. Additionally, one could infer from the table that since CTP enables CFO and CFO enables CTP, these two transformations should be applied repeatedly before LUR. Of course, there may be other factors to consider when applying loop unrolling. In this paper, we focus on only one, namely transformation interactions. Other factors may include the impact that the unrolled loop has on the cache. When other factors are important in the application of transformations, these factors could be embedded in Genesis experiments (e.g., by adding measures of cache performance).

Table 3 also displays the interactions that disable LUR. As FUS is the only transformation that disables LUR, a decision must be made about the importance of applying FUS on the target architecture. If LUR is more important, then either FUS should not be applied at all or only at the end of the transformation process.

The information about interactions could also be used in the development of a transformation guidance system that informs the user when a transformation has the potential for disabling another transformation and also informs the user when a transformation has the potential for enabling another transformation. The interactions among the transformations can also be used to determine some pairwise orderings of transformations. For instance, Table 3 indicates that when applying CPP and CTP, CPP should be applied first. Other such information can be gleaned from this table.

4.2 Genesis: An Automatic Transformer Generator Tool

Another use of the framework is the construction of a transformer tool that automatically produces transformation code for the specified transformations. The Genesis tool analyzes a Gospel specification and generates code to perform the appropriate pattern matching, check for the required data dependences, and call the necessary primitive routines to apply the specified transformation.[19] Figure 6 presents a pictorial description of the design of Genesis. The value of Genesis is that it greatly reduces the programmer's burden by automatically generating code rather than having the programmer implement the optimizer by hand. In Figure 6, a code transformer is developed from a generator and constructor.

The generator produces code for the specified transformations, utilizing pre-defined routines in the transformer library, including routines to compute data and control dependencies. The constructor packages all of the code produced by the generator, the library routines, and adds an

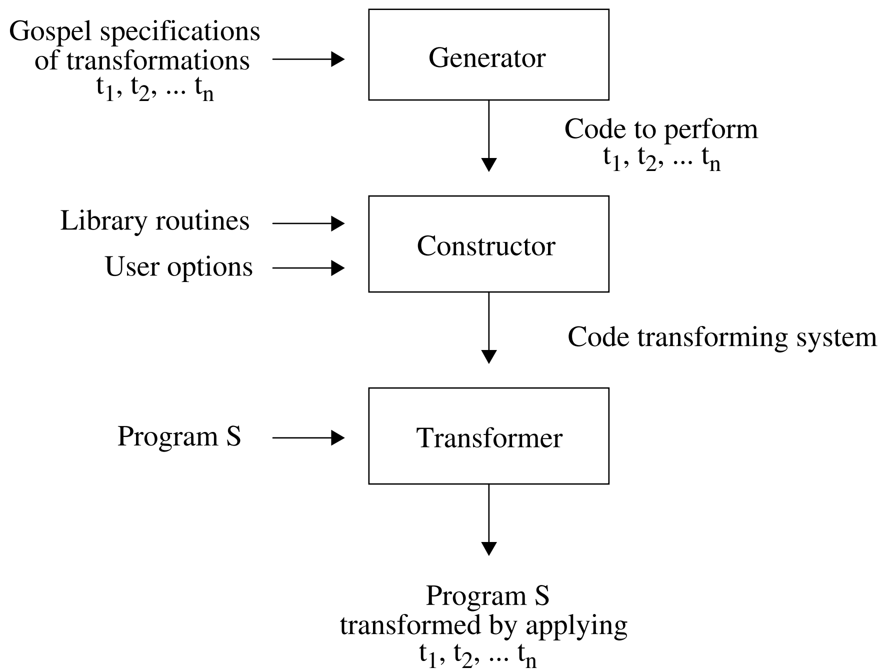

Figure 6. Overview of Genesis

interface which prompts interaction with the user. The generator section of Genesis analyzes the Gospel specifications using LEX and YACC, producing the data structures and code for each of the three major sections of a Gospel specification. The generator 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 three address code. Code to verify the required data dependences is next generated. Finally, code is generated for the action statements. The Genesis system is about 6,500 lines of C code, which does not include the code to compute data dependencies. A high level representation of the algorithm used in Genesis is given in Figure 7.

The generated code relies on a set of predefined routines found in the transformer library. These routines are transformation independent and represent routines typically needed to perform transformations. The library contains pattern matching routines, data dependence computation algorithms, data dependence verification procedures, and code manipulation routines. The pattern matching routines search for loops and statements. Once a possible pattern is found, the generated code is called to verify 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

```
GENESIS()
  For I = 1 to n          {iterate through transformation list}
      Read(Gospel specification for Transformation t_i)
      {Analyze the Gospel specifications using LEX and YACC}
      gen_code_structures(type_section_declarations)  {Gen code to setup data structures}
      gen_code_pattern_search(code_pattern) {Gen code to search for patterns}
      gen_code_depend_verify(data_dependences) {Gen code to verify data dependences}
      gen_code_actions(primitive_operations) {Gen code to perform primitive actions}
  End for
      {Create the interface from a template}
  Construct_optimizer(generated_code, library_routines)
  Read(source_code)
  Convert( source, intermediate representation)
  While (user_interaction_desired)
      Select_transformations
      Select_application_points
      Override(dependence_restrictions) {optional}
      Compute_data_dependences
      Perform_optimization (user's_direction)
  EndWhile
```

Figure 7. The Genesis Algorithm

loop or set. The generated code may simply be an "if" to ensure a dependence does not exist or may be a more complex integration of tests and loops. For example, if all statements dependent on S_i 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 routines from the transformer library and the generated code to produce the transformer for the set of transformations specified. The constructor also generates an interface to execute the various transformations. The interface to the transformer 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 transformations in any order. The user may elect to perform a transformation at one application point (possibly overriding dependence constraints) or at all possible points in the program.

4.2.1. Prototype Implementation

In order to test the viability and robustness of this approach, we implemented a prototype for Genesis and produced a number of transformers. For ease of experimentation, our prototype produces a transformer for every transformation specified.

For any transformation specified, the generator produces four procedures tailored to a transformation: *set_up_Trans, match_Trans, pre_Trans,* and *act_Trans*. These procedures correspond to the DECLARATION, Code_Pattern, Depend and ACTION sections in the specifications.

In our implementation, a transformer consists of a driver that calls the routines that have been generated specifically for that transformation. Code for the driver is given in Figure 8. The format of the driver is the same for any transformer generated. The driver calls procedures in the generated call interface for the specific transformation *(set_up_Trans, match_Trans, pre_Trans, and act_Trans)*. The call interface in turn calls the generated procedures that implement the transformation (the generated transformation specific code). For CTP, as given in Figure 9, the *set_up_Trans* 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_Trans* and *pre_Trans* of the call interface procedures return a boolean value.

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

Figure 8. The Driver Algorithm

Any generated set_up procedure consists of code that initializes data structures for each element specified using **any** or **all** in the PRECONDITION section. A type table data structure, TypeTable, contains identifying information about each statement or loop variable specified in the DECLARATION section. The TypeTable holds the identifier string, creates an entry for a quantifier that may be used with this identifier in the PRECOND section, and maintains the type of the identifier (e.g., statement, loop, adjacent loop or nested loop). For type Statement, an entry is initialized with the type and corresponding identifier. If a loop-typed variable is specified, additional flags for nested or adjacent loops are set in the type table entry. These entries are filled in as the information relevant to the element is found when the transformations are performed. For each statement in the DECLARATION section, a call to TypeTable_Insert is generated with the identifier and the type of the identifier and placed in the set_up procedure. During execution of CTP, shown in Figure 9, a type table entry is initialized with type "Statement" and identifier S_i when the transformer executes procedure *set_up_CTP*.

After the *set_up_CTP* procedure terminates, the driver indirectly initiates an exhaustive search for the statement recorded in the type table by calling *match_CTP*. If the source program's statement does not match, then the transformer driver re-starts the search for a new statement. The match procedure is generated from the statements in the Code_Pattern section of the Gospel specification. For each quantified statement in the Code_Pattern section, a call to SetTable_Insert is made with the identifier, type of identifier, and quantifier. SetTable_Insert searches for the requested type and initializes the Set_Table data structure with the appropriate attributes for the type (e.g., for a statement, the opcode and operands are set). Next the restrictions in the Code_Pattern section are directly translated into conditions of IF statements to determine if the requested restrictions are met. If the current quantifier is an "all", then a loop is generated to check all of the objects found by Set_Table. In the CTP example in Figure 9, code is generated that searches for an assignment statement with a constant on the right hand side.

The next routine is the pre procedure, which is generated from the statements in the Depend section. For each quantified statement, a call to SetTable_Insert is generated (however, the pattern matching will not be performed again at run-time.) For the CTP example, the *pre_CTP* procedure inserts an element into the Set_Table structure for each dependence condition statement. S_j is inserted into Set_Table and the dependence library routine is called to find the first statement that is flow dependent on S_i; if no statement is found then the condition fails. S_l is also inserted into the Set_Table and the dependence routine is called again. Each S_l such that S_l is flow dependent on S_j is examined to determine if the operand of S_l causing the dependence is the same variable involved in the dependence from S_i to S_j. If such an S_l is found then the condition fails. Next, an assignment statement is generated to assign the "hits" field of the Set_Table data structure with the result of the requested dependence or membership procedure call. For example, by setting the "hits" field to a result of a flow_dependence call, the hits field will contain either 1 (for the any quantifier) or many

```
set_up_CTP(){
    TypeTable_Insert(Statement, Si);                    set up type table for statement Si
}

match_CTP() {
    Set Table_Insert(Statement, Si, any);               classify Si as a set of statements
    if (SetTable[Si].opcode.kind != ASSGN) then
        return (failure);                               if Sj's opcode is not ASSGN, fail
    if (setTable[Si].operand_a.kind!= CONST) then
        return (failure);
    return(success);                                    match successful for Si
}

pre_CTP() {
    SetTable_Insert(Statement, Sj, all);                classify Sj as a set of statements
    SetTable_Insert(Statement, Sl, no);                 classify Sl as a set of statements

    SetTable[Si].hits = flow_dep(Si,any, EQ);           find and assign flow dep Sj
    If(SetTable[Si].hits ==NULL) then                   if flow dep Sj does not exist try again
        return (failure);

    SetTable[Sl].hits = anti_dep(Sj,any,EQ);            find anti dep of Sj
    foreach(SetTable[Sl].hits {
        if((Si.quad_num) != (Sl.quad_num)               compare quad_numbers and
            &&dep_pos(Sj)) ==dep_pos(Sl)))              operand involved in dependencies
                return(failure);
    return(success);
}

act_CTP() {                                             modify one of Sj's operands

    if(Si.oprc==Sj.orpa) then
        modify (Sj.opra, Si.oprc);
    else
        modify(Sj.oprb,Si.oprc);
    endif
    return(success);
}
```

Figure 9. The Generated Code for CTP

(for the all quantifier) statement numbers that are flow dependent with the required direction vector. Next, IF statements are directly generated from any relational conditions that exist in the specification.

The last procedure to be called is the action procedure. The action procedure is generated from the statements in the Action section of the Gospel specification. For each individual action, a call to the primitive transformation is made with the required parameters (e.g., modify requires the object being modified and the new value). If the Gospel forall construct is used, then a for loop is generated and the calls to the primitive transformations are placed within the loop. In the example in Figure 9, *act_CTP* simply modifies the operand collected in S_j. This modification occurs in either the first or second modify statement depending on the operand that carries the dependence. Thus, the first call to *modify* considers “operand a” of S_j 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. The procedure *act_CTP* is called by the driver only if *match_CTP* and *pre_CTP* have terminated successfully. For more implementation details, the reader is referred to another paper.[18]

5. Experimentation

Using our prototype implementation of Genesis, we performed experiments to demonstrate that Genesis can be used to explore the properties of transformations including 1) the frequency of applying transformations, and 2) the interactions that occur among the transformations.

Using Genesis, transformers were produced for ten of the twenty transformations specified: CPP, CTP, DCE, ICM, INX, CRC (Loop Circulation), BMP (Bumping), PAR (Parallelization), LUR, and FUS. Experimentation was performed using programs found in the HOMPACK test suite and in a numerical analysis test suite.[2] A short description and the Gospel specifications of these transformations are given in Appendix B. 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 programs to solve non-linear equations using Newton's method. A total of ten programs were used in the experimentation. The benchmark programs were coded in Fortran, which was the language accepted by our front end. They ranged in size from 110 to 900 lines of intermediate code statements. The programs were numerical in nature and had a mixture of loop structures, including nested, adjacent and single loops. Both traditional optimizations and parallelizing transformations could be applied in the programs, as we were interested in the interaction between these types of transformations. Longer programs would more likely show more opportunities for transformations and thus more opportunities for interactions.

In order to verify Genesis’ capability to find application points, four transformations were specified in Gospel and run on the HOMPACK test suite. The number of application points for each of the transformations was recorded and compared to the number of application points found by Tiny.[20] The comparison revealed that the Genesis found the same number of applications points that Tiny found. Furthermore, seven optimizations were specified in Gospel and optimizers were generated by Genesis. The generated optimizers were compared to a hand-coded optimizer to further verify Genesis’ ability to find application points. Again, the optimizers generated by Genesis found the same application points for optimizations.

In the test programs, CTP was the most frequently applicable transformation (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 transformations, 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 transformation. These results are shown in Table 4 where a "-" entry indicates that no interaction is theoretically possible and a number gives the number of interactions that occurred. For example, the entry for INX/FUS indicates that 5 enabling interactions were found and 4 disabling interactions were found in the 13 application points.

	Freq	**DCE**	**CTP**	**CPP**	**CFO**	**ICM**	**LUR**	**FUS**	**INX**
DCE	34	0/-	-/2	0/0	-/-	0/-	-/-	0/-	0/-
CTP	97	13/-	-/-	-/0	5/-	0/-	41/-	0/-	0/-
CPP	5	-/-	0/-	-/-	-/-	0/0	-/-	0/-	-/-
CFO	5	-/-	41/-	-/-	-/-	-/-	0/-	0/-	-/-
ICM	0	-/-	-/-	-/-	-/-	0/-	-/-	0/0	0/0
LUR	49	0/-	10/-	0/-	0/-	-/-	0/-	1/7	2/6
FUS	11	-/-	-/5	-/-	-/-	-/0	-/1	1/0	0/6
INX	13	-/-	-/-	-/-	-/-	0/0	-/-	5/4	4/0

Table 4. Enabling and Disabling Interactions

To investigate the ordering of transformations, we considered the transformations FUS, INX and LUR which we showed in Section 4 to theoretically enable and disable one another. In one program, FUS, INX, and LUR were all applicable and heavily interacted with one another by creating and destroying opportunities for further transformations. For example, applying FUS disabled INX and applying LUR disabled FUS. Different orderings produced different transformed programs. The transformations also interacted when all three transformations 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 a transformation at some point in the program may prevent another transformation 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 transformations as a guide, the user may need multiple passes to discover the series of transformations that would be most fruitful for a given system.

The framework could also be used to explore the value of combining transformations.

Blocking is a transformation that combines Strip Mining and Interchange.[11] We performed a preliminary experiment in which we applied various orders of Loop Interchange (INX), Loop Unrolling (LUR) and Loop Fusion (FUS). In the experiments, LUR when followed by INX produced more opportunities for transformations than other orders. Thus, after performing experimentation to examine what happens when a series of transformations are applied, it might be beneficial to combine certain transformations and apply them as a pair. In our example, we would consider combining LUR and INX.

6. Concluding Remarks

The code improving transformation framework presented in this paper permits the uniform specification of code improving transformations. The specifications developed can be used for analysis and to automatically generate a transformer. The analysis of transformations enables the examination of properties such as how transformations interact to determine if a transformation creates or destroys conditions for another transformation. These relationships offer one approach for determining an order in which to apply transformations to maximize their effects. The implementation of the Gospel specifications permits the automatic generation of a transformer. Such an automated method enables the user to experimentally investigate properties by rapidly creating prototypes of transformers to test their feasibility on a particular machine. Genesis also permits the user to specify new transformations and quickly implement them.

Future work in this research includes examining the possibility of automatically proving interactions by expanding the specifications to a more detailed level. Such a transformation interaction proving tool would enable the user to determine properties of the transformations. Also, the design of a transformation guidance system prototype is being examined for its feasibility. This type of system would aide the user in applying transformations by interactively providing interaction information. The Gospel specifications are also being explored to determine if they can easily be combined to create more useful transformations.

Acknowledgment:

We are especially grateful to TOPLAS Associate Editor Jack Davidson for his insightful criticisms and advice on earlier drafts of this paper. We also thank the anonymous referees for their helpful comments and suggestions, which resulted in an improved presentation of the paper.

Appendix A

PRECONDITION 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
```

Appendix B

Gospel Specification of Transformations

Bumping (BMP): Modify the loop iterations by bumping the index by a preset amount (e.g., 2).

```
DECLARATION
        Statement: S, S_new;
        Loop L;
PRECONDITION
  Code_Pattern
        any L;
  Depend
        all S: flow_dep (L.Lcv, S, (any));
ACTION
        forall S
        add (S.Prev, ( -, 2, S.opr_1, S.opr_1), S_new);
        modify (L.Initial, eval(L.Initial, +, 2));
        modify (L.Final, eval(L.Final, +, 2));
```

Constant Folding (CFO): Replace mathematical expressions involving constants with their equivalent value.

```
DECLARATION
        Statement: S_i;
PRECONDITION
  Code_Pattern                    Find a constant expression
        any S_i: type(S_i.opr_2) == const
                AND type(S_i.opr_3) == const AND S_i.opcode != assign;
  Depend                          No Dependence checks
ACTION                            Fold the constants into an expression
        modify (S_i.opr_2, eval(S_i.opr_2, S_i.opcode, S_i.opr_3);
        modify (S_i.opcode, assign);
```

Copy Propagation (CPP): Replace the copy of a variable with the original.

```
DECLARATION
        Statement: S_j, S_i, S_k, S_p;
PRECONDITION
  Code_Pattern                    find a copy statement
        any S_i: S_i.opcode == assign
```

AND type (S_i.opr$_2$) == var AND type (S_i.opr$_1$) == var;

Depend *all uses do not have other defs along the path*

all (S_j, pos): flow_dep (S_i, S_j, (=));

no S_k: flow_dep (S_k, S_j, (=)) AND (S_k != S_i);

no S_p: mem (S_p, path (S_i, S_j)), anti_dep (S_i, S_p, (=));

ACTION *propagate and delete the copy*

forall S_j

modify (operand (S_j, pos), S_i.opr$_2$);

delete (S_i);

Loop Circulation (CRC):Interchange perfectly nested loops (more than two)

DECLARATION

Statement: S_n, S_m;

Tight: (L_1, L_2, L_3);

PRECONDITION

Code_Pattern *Find Tightly nested loops*

any (L_1, L_2, L_3);

Depend *Ensure perfect nesting, no flow_dep with <,>*

no L_1.Head: flow_dep (L_1.Head, L_2.Head);

no L_2.Head: flow_dep (L_2.Head, L_3.Head);

no S_m, S_n: mem (S_m, L_3) AND mem (S_n, L_3),

flow_dep (S_n, S_m, (<,>));

ACTION *Interchange the loops*

move (L_1.Head, L_3.Head);

move (L_1.End, L_3.End.Prev);

Common Sub-Expression Elimination (CSE): Replace duplicate expressions so that calculations are perfomed only once.

DECLARATION

Statement: S_n, S_m, S_k, S_j;

PRECONDITION

Code_Pattern *Find binary operation*

any S_n: S_n.opcode == + OR S_n.opcode == - OR S_n.opcode == * OR S_n.opcode == /;

Depend *Find common sub-expression*

any S_m: S_m.opcode == S_n.opcode AND S_m.opr$_2$ == S_n.opr$_2$ AND S_m.opr$_3$ == S_n.opr$_3$;

no (S_k, pos): anti_dep(S_n, operand(S_k, pos)) AND flow_dep (operand(S_k,pos), S_m);

all S_j: ctrl_dep(S_j, S_n) AND ctrl_dep(S_j, S_m);

ACTION

add ((S_n.opcode, temp, S_n.opr$_2$, S_n.opr$_3$), ,S_n)

modify (S_n, (assign, S_n.opr$_1$, temp)

modify (S_m, (assign, S_m.opr$_1$, temp)

Dead Code Elimination (DCE): Remove statements that define values for variables that are not used.

DECLARATION

Statement: S_i, S_j;

PRECONDITION

Code_Pattern *find statement assigning variable, value or expression*

any S_i: S_i.opcode == assign OR S_i.opcode == - OR S_i.opcode == *

OR S_i.opcode == + OR S_i.opcode == /;

Depend *statement may not be used*

no S_j: flow_dep (S_i, S_j, (any));

ACTION *delete the dead code*

delete (S_i);

Loop Fusion (FUS): Combine loops with the same headers.

DECLARATION

Statement: S_n, S_m, S_i, S_j, S_k;

Adjacent: (L_1, L_2);

PRECONDITION

Code_Pattern *find adjacent loops with equivalent Heads*

any L_1, L_2: L_1.Initial == L_2.Initial

AND L_1.Final == L_2.Final

AND L_1.Lcv == L_2.Lcv

Depend *no dependence with backward direction first; no def reaching prior to loops*

no S_n, S_m: mem (S_n, L_1) AND mem (S_m, L_2),

flow_dep (S_n, S_m, (=*, >, any)) OR out_dep (S_n, S_m, (=*, >, any)) OR

anti_dep (S_n, S_m, (=*, >, any));

no S_i, S_j: mem (S_j, L_1) AND mem (S_k, L_2),

flow_dep (S_i, S_j, (any)) AND anti_dep (S_j, S_k, (any)) AND (S_i != S_k);

ACTION *Fuse the loops*

modify (L_1.Head.opr$_1$, L_2.Head.Label);

modify (L_2.End.opr_1, L_1.End.Label);

delete (L_1.End);

delete (L_2.Head);

Invariant Code Motion (ICM): Remove statements from within loops where the values computed do not change.

DECLARATION

Statement: S_k, S_m;

Loop: L;

PRECONDITION

Code_Pattern *any loop*

any L;

Depend *any statement without dependence within the loop*

any S_k: mem (S_k, L) AND mem (S_m, L),

NOT (flow_dep (S_m, S_k) OR anti_dep (S_k, S_m) OR

((out_dep (S_m, S_k) OR out_dep (S_k, S_m)) AND (S_m != S_k)

OR anti_dep (S_m, S_k) OR ctrl_dep (S_m, S_k) OR flow_dep(S_k, S_k)

OR anti_dep (S_k, S_k) OR flow_dep (L, S_k)));

ACTION *move statement to within header*

move (S_k, L.Start.Prev);

Loop Unrolling (LUR): Duplicate the body of a loop.

DECLARATION

Statement: S_i, S_j;

Loop: L_1, L_2;

PRECONDITION

Code_Pattern *any loop iterated at least once*

any L_1: type (L_1.Initial) == const AND type (L_1.Final) == const

AND L_1.Final - L_1.Initial > 0;

Depend *No Dependence checks*

ACTION *unroll one iteration, update original loop's Initial*

copy (L_1.Body, L_1.Head.Prev, L_2);

modify (L_2.Final, L_2.Initial);

modify (L_1.Initial, eval(L_1.Initial, +, 1));

delete (L_2.End);

delete (L_2.Head.Label);

Parallelization (PAR): Modify loop type for parallelization.

DECLARATION

Statement: S_i, S_j;

Loop: L_1;

PRECONDITION

Code_Pattern

any L_1: L_1.opcode == DO;

Depend

no S_i, S_j: mem (S_i, L_1) AND mem (S_j, L_1), flow_dep(S_i, S_j, (=, *));

ACTION

modify (L_1.opcode, PAR);

Strip Mining (SMI): Modify loop to utilize vector architecture.

DECLARATION

Loop: L;

PRECONDITION

Code_Pattern

any L: L.Final - L.Initial > SZ;

Depend

ACTION

copy (L.Head, L.Head.Prev, L_2.Head);

modify (L_2.Lcv, temp(T));

modify (L_2.step, SZ);

modify (L_1.Initial, T);

modify (L_1.Final, MIN(T + SZ- 1, L_1.Final));

copy (L.End, L.End, L_2.End);

modify(L_2.End, address(L_2.Head));

Loop Unswitching (UNS): Modify a loop that contains an IF to an IF that contains a loop.

DECLARATION

Loop: L;

Statement: S_i, S_k;

PRECONDITION

```
Code_Pattern
        any L;
Depend
        any S_i: mem (S_i, L), S_i.opcode == IF AND ctrl_dep (S_i, L.End.Prev.Prev)
                AND NOT flow_dep(L.Head, S_i);
        Find the Else
        any S_k: mem (S_k, L) AND NOT ctrl_dep(S_i, S_k);
ACTION
        copy (L.Head, S_k, L_2.Head);
        copy (L.End, L.End.Prev.Prev, L_2.End);
        modify (L_2.End, address(L_2.Head));
        move (L.Head, S_i);
        move (L.End, S_k.Prev);
```

REFERENCES

1. M. Alt and F. Martin, "Generation of Efficient Interprocedural Analyzers with PAG," A. Mycroft, Ed., *Lecture Notes in Computer Science,* Springer, Heidelberg. 1997.
2. Richard Burden and J. Douglas Faires, in *Numerical Analysis*, Prindle, Weber & Schmidt, Boston, MA, 1989.
3. Clifford Click, "Global Code Motion Global Value Numbering," *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pp. 246-257, June, 1995.
4. Jack W. Davidson and Christopher W. Fraser, "Automatic Generation of Peephole Transformations," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 111-115, 1984.
5. Matthew Dwyer and Lori Clarke, "A Flexible Architecture for Building Data Flow Analyzers," *Proceedings of the 18th International conference on Software Engineering*, Berlin, Germany, pp. 554-564, March, 1996.
6. Christopher W. Fraser and Alan L. Wendt, "Automatic Generation of Fast Optimizing Code Generators," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 79-84, June, 1988.
7. Robert Giegerich, "Automatic Generation of Machine Specific Code Transformer," *Proceedings of Ninth Annual ACM Symposium on Principles of Programming Languages,* pp. 75-81, Jan., 1982.
8. *GNU C Compiler Manual* (V.2.7.2), Free Software Foundation, Inc., Boston, 1995.
9. Robert R. Kessler, "Peep - An Architectural Description Driven Peephole Transformer," *Proceedings of the ACM SIGPLAN '84 Symposium of Compiler Construction, SIGPLAN Notices,* Vol. 19, no. 6, pp. 106-110, June 1984.
10. David A. Padua and Michael J. Wolfe, "Advanced Compiler Transformations for Supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184-1201, Dec., 1986.
11. Vivek Sarkar and Radhika Thekkath, "A General Framework for Iteration-Reordering Loop Transformations," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 175-187, June, 1992.
12. Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994, Version 1.0.
13. Steven Tjiang and John Hennessy, "Sharlit - A tool for building transformers," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 82-93, June, 1992
14. G. A. Venkatesh and Charles N. Fischer, "SPARE: A Development Environment for Program Analysis Algorithms," *IEEE Transactions of Software Engineering*, Vol. 18, No. 4, April, 1992, pp. 304-318.
15. Tia Watts, Mary Lou Soffa and Rajiv Gupta, "Techniques for Integrating Parallelizing Transformations and Compiler Based Scheduling Methods," *Proceedings of Supercomputing '92*, Minneapolis, MN, ,pp. 830-839, Nov. 1992.
16. Deborah Whitfield and Mary Lou Soffa, "An Approach to Ordering Optimizing Transformations," *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, pp. 137-146, March, 1990.

17. Deborah Whitfield and Mary Lou Sofa, “Investigation of Properties of Code Transformations,” *Proceedings of 1993 International Conference on Parallel Processing*, St. Charles, Illinois, pp. 156- 160, August, 1993.

18. Deborah Whitfield and Mary Lou Soffa, “The Design and Implementation of Genesis,” *IEEE Journal of Software Practices and Experiences*, Vol. 24(3), pp. 307-325, March, 1994.

19. Deborah Whitfield and Mary Lou Soffa, “Automatic Generation of Global Optimizers,” *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* pp. 120-129, June, 1991.

20. Michael Wolfe, *Tiny: A Loop Restructuring Research Tool*, Oregon Graduate Institute of Science and Technology, 1989.

21. Michael Wolfe, in *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood, CA, 1996.