Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

A Unifying Framework for Undoing Code Transformations

Chyi-Ren Dow
crdow@iecs.fcu.edu.tw
Department of Information Engineering
Feng-Chia University
Taichung, Taiwan 407
R. O. C.

Mary Lou Soffa and Shi-Kuo Chang
soffa@cs.pitt.edu and chang@cs.pitt.edu
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
U. S. A.

Abstract

A transformation applied to optimize or parallelize a program may be found to be ineffective, or may be made invalid by code changes. In this paper, we present a unifying framework to remove such transformations. The technique employs inverse primitive actions, making it transformation independent. The order of undoing the transformations is independent of the application order. The technique uses post conditions of a transformation to determine whether the transformation can be immediately removed. An undo algorithm and the analysis for correctness of the algorithm are presented. An undo facility has been implemented and experimental results for applicable transformations and enabled transformations are reported.

1. Introduction

A transformation applied to optimize or parallelize a program may be found to be ineffective, or may be made invalid by edits to the code. Although numerous parallelizing compilers and parallelization systems have been designed and implemented [2, 8, 11], no undo facility, a very important facility in interactive environments, is supported in these systems.

In an approach to incremental reoptimization of programs, a scheme has been developed to remove traditional optimizations when a program is modified by edits [10]. It works on intermediate-level program representations, namely a control flow graph and dag representation, with history information of optimizations placed on dag nodes. Problems in extending this scheme for parallelizing transformations include (1) the need of a representation that can support the application of scalar and parallelizing transformations to code elements in different levels such as loops, statements, and operand expressions, (2) the need for different information stored not only for eliminated, relocated, and replaced statements but also for restructured loop structures and duplicated statements, and (3) the desire of a transformation independent technique since new transformations may be developed and incorporated into the system.

This research approaches these problems by the development of a unifying framework for undoing code transformations. The techniques employ inverse primitive actions to undo transformations, making the techniques transformation independent. Pre and post conditions of transformations are utilized to determine whether an applied transformation remains safe and whether it is immediately reversible. Due to transformation interactions, a transformation may enable the application of a transformation or disable a transformation. Therefore to remove a transformation, it may be necessary to first remove affecting transformations that were applied after the transformation. After a transformation has been removed, disabled transformations would have to be removed also.

The remainder of this paper is organized as follows. In Section 2, we discuss a scheme developed to apply code transformations and information to be stored in order to allow the reversal of transformations. Section 3 describes the interactions of code transformations, the pre and post conditions of transformations and an undo algorithm. The analysis of the undo algorithm is presented in Section 4. The implementation of transformation application and undo facilities and experimental studies of applicable and enabled transformations are described in Section 5. Finally, we discuss incremental transformations when a program is modified by edits and conclusions are drawn in Section 6.

2. Applying Code Transformations

Traditional optimizations are generally applied on intermediate scalar code and rely on data-flow information. Parallelizing transformations, which are applied to source code to exploit parallelism, involve the transformation of loops and rely on data dependence information. In what follows, we describe the 7 representative transformations, including traditional and parallelizing transformations, that are used as the test set in this paper.

Dead Code Elimination (DCE) [1,3]: deletes a statement that defines a variable that is never used.
Constant Propagation (CTP) [1]: propagates a constant to a single use.
Invariant Code Motion (ICM) [1,3]: moves loop invariant code outside of the loop.
Loop Unrolling (LUR) [3]: unrolls the body of a loop at least once.
Strip Mining (SMI) [9, 14]: transforms a single nested loop into a doubly nested one.
Loop Fusion (FUS) [3]: converts two adjacent loops into one loop.
Loop Interchanging (INX) [4, 14]: interchanges two tightly nested loops.

Our approach to performing transformations and undoing transformations is by using primitive actions [6]. The actions of applying transformations can be described by a sequence of primitive operations. The five primitive operations are listed below. These operations are overloaded in that they can apply to different types of code elements such as loops, statements, nested loops, operands, etc. In the following descriptions, *a*, *b* and *c* refer to any type of code element. The five actions are:

Delete (*a*): delete *a*.
Copy (*a*, *b*, *c*): copy *a*, name it *c*, and place it following *b*.
Move (*a*, *b*): remove *a* from its original position and place it following *b*.
Add (*a*, Element_description, *b*): add an element described by Element_description, call it *b*, and place it following *a*.
Modify (Operand(S,i), New_operand): modify Operand i of statement S to be New_operand.

In order to allow the reversal of transformations in any order, sufficient information must be recorded to keep a history of all existing transformations. Information is required to ensure correct detection of invalidated transformations and the removal of invalid transformations. Our approach is to store information about code patterns before and after the application of a transformation as well as a sequence of primitive actions that accomplishes the transformation. The history of applied transformations is maintained on the program representation given by transformation independent annotations. A pre_pattern notation is used to determine whether the transformation is applicable and a post_pattern is used to determine whether the transformation is immediately reversible as discussed in the following sections. For example, the pre_pattern for DCE is a pointer to the statement to be deleted, the primitive action is a Delete operation, and the post_pattern is a pointer to the deleted code and a pointer to the original location of the dead code which is saved for possible later restoration. It should be noted that the information of the pre_pattern and primitive actions of a transformation can be obtained automatically if the approach taken in the development of a code transformation is to specify the transformation using primitive transformations and let the transformation generator automatically generate the transformation from the specification. Intuitively, the post pattern can be obtained automatically given the pre_pattern and primitive actions of a transformation since the post_pattern results from applying the primitive actions on the pre_pattern.

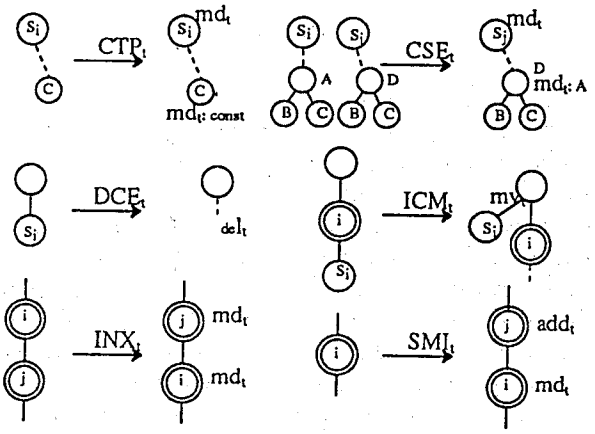

Figure 1. Annotations based on primitive actions.

In order to maintain a complete snapshot of existing transformations, adequate to determine when a transformation becomes unsafe or whether it is immediately reversible, appropriate transformation history is annotated on our program representation. Our two-level representation integrates two program representations, a high level one - an Augmented PDG (APDG) and a low-level one Augmented DAG (ADAG) to allow the application of both parallelizing transformations and traditional optimizations. Since it is desirable to have transformation independent annotations, our annotations of a transformation on the representations are based on the primitive actions and an order stamp (t) associated with the transformation. The order stamp is used to link the primitive action with the transformation that caused it and is used to determine whether a transformation may be affected when undoing a transformation in an independent order. Figure 1 shows the transformation annotations for the transformations applied in which md, mv, cp, and del are abbreviation of modify, move, copy, and delete.

Figure 2 shows a transformation application algorithm. The transformation history of code patterns before and after the application of the transformation, applied primitive actions, and the transformation independent

annotations are stored to enable the removal of transformation in an order independent of application order.

```
1  Procedure APPLY(t_i)
2  BEGIN
3   Set up data structures for code elements of t_i;
4   match_suc := search for the code patterns of t_i;
5   while(match_suc)
6   THEN BEGIN
7     pre_suc := verify data dependences of t_i;
8     if(pre_suc)
9     THEN BEGIN
10      Store the pre-pattern of t_i
11      For each primitive action a_ij of t_i
12       DO BEGIN
13         Store a_ij;
14         Store transformation annotations on the representation;
15         Perform the primitive action a_ij;
16       END {DO}
17      Store the post-pattern of t_i
18      Return(1);
19    END {THEN}
20    match_suc := Search for the code patterns of t_i;
21  END {WHILE}
22  Return(0);
23 END
```

Figure 2. A transformation application algorithm.

3. Undoing Code Transformations

Due to the complexity of transformation interactions (transformation enabling and disabling), the reversal of transformations in an order independent of their application order may not be possible or safe by directly performing the inverse actions of the transformations. Therefore to remove a transformation, it may be necessary to first remove affecting transformations. After a transformation has been removed, disabled transformations would have to be removed also. This section describes the interactions of code transformations, the pre and post conditions of transformations and our undo algorithm.

3.1. Interactions of Code Transformations

Interactions of code transformations may occur by one transformation enabling the application of another transformation that previously could not be applied, or one transformation disabling conditions that exist for another transformation [10, 12]. If conditions of a transformation t_j are enabled or disabled by another transformation t_i, t_i is defined to be an *affecting transformation* to t_j and t_j is defined as an *affected transformation* by t_i. Enabling interactions among transformations occur when performing a transformation enables conditions for other transformations to become applicable. Since dependencies established by chains of enabling interactions yield similar chains of disabling interactions when a transformation is destroyed, rippling effects are caused in that removing a transformation destroys the safety of other transformations that must also be removed.

Table 1 shows the enabling interactions of a set of code transformations, which include traditional optimizations and parallelizing transformations. A "x" entry in a particular row and column denotes that the transformation in that row enables the transformation in that column. Since enabling interactions are perform-create dependencies and dependencies established by chains of creations yield similar chains of destruction when a transformation is destroyed, the reverse-destroy dependencies exactly replicate the perform-create dependencies [10]. Thus, the reverse-destroy table can be used in a heuristic to reverse code transformations. When a transformation is reversed, only transformations with a mark "x" in the reverse-destroy table need to be considered as possibly affected transformations.

3.2. Pre and Post Conditions of Transformations

Pre-conditions of a transformation are conditions that must exist before the application of the transformation. The satisfaction of pre-conditions determines whether the transformation is safe to apply. The existence of safety disabling conditions determines whether the transformation remains safe. Post conditions are conditions that result from applying the transformation. Post conditions of a transformation can be altered by applying and removing transformations. Two kinds of transformation conditions are discussed in this section. One is the safety condition that is used to determine whether an applied transformation remains safe. The other is the reversibility condition that considers whether a transformation is immediately reversible by directly performing its inverse actions.

(1) **Safety:** A transformation is **safe** if it preserves the meaning of a source program. The safety conditions of a transformation t_i can be altered when a transformation t_j, applied before t_i, is reversed. Removing a transformation may destroy the safety of another transformation but performing a transformation can never destroy the safety of already applied transformations because a transformation is never performed on the premise that another transformation will be reversed to make it safe [10]. Therefore, given a sequence of transformations, $T = \{t_1, t_2, ..., t_n\}$, the safety of transformation t_i can be disabled by the reversal of a preceding transformation t_j, where $1 <= j < i$. As shown in Table 2, which gives pre-conditions, safety-disabling conditions, and reversibility-disabling conditions for a set of transformations, the safety-disabling conditions of a transformation are determined by negating the pre-condition of a transformation.

	DCE	CSE	CTP	CPP	CFO	ICM	LUR	SMI	FUS	INX
DCE	x	x	-	x	-	x	-	-	x	x
CTP	x	x	-	-	x	x	-	x	x	x
ICM	-	x	-	-	-	x	-	-	x	x
LUR	x	x	x	x	x	x	x	x	x	-
SMI	-	-	-	-	-	-	-	-	x	-
FUS	-	-	-	-	-	-	-	-	x	x
INX	-	-	-	-	-	x	-	-	x	x

Table 1. Perform-create (reverse-destroy) interactions.

Transformation	Disabling Conditions of Safety	Disabling Conditions of Reversibility
Dead Code Elimination (DCE) *Pre-condition*: $\exists$ $S_i \wedge \nexists$ $S_l \ni (S_i \delta_= S_l)$.	1) $\exists$ $S_l \ni (S_i \delta S_l)$: • Add a statement S_l that uses value computed by S_i. • Modify a statement S_l that uses value computed by S_i. • Move a statement S_l on the path so that S_i reaches. †	1) The original location of S_i cannot be determined: • Delete context of the location. (e.g., delete the loop it belongs to) • Copy context of the location. (e.g., copy the loop it belongs to by LUR)
Loop Unrolling (LUR) *Pre-condition*: $\exists$ type(L.initial) == const $\wedge$ type(L.final) == const $\wedge$ L.final - L.initial > 0.	1) $\nexists$ DO loop, L: • Remove loop L.	1) $\nexists$ the unrolled loop: • Delete L 2) the loop expression changed: • Modify(loop_exp(L)) (e.g., by LUR, INX, or SMI) 3) $\nexists$ unrolled statement. • Delete S_i.
Loop Interchanging (INX) *Pre-condition*: $\exists$ tight loops (L_1, L_2) $\wedge$ $\nexists$ $S_n \in L_2 \ni \exists$ $S_m \in L_2 \ni (S_n \delta_{<,>} S_m)$ $\wedge$ NOT (L_1.head $\delta_=$ L_2.head)	1) $\nexists$ two nested DO loops: • Insert a statement between the two loops. • Delete a loop. 2) $\exists$ two statements, S_n and S_m such that there is a (<,>) dependence vector: • Insert S_n. † • Insert S_m. • Remove a definition so that the dependence holds. † • Create a path between S_n and S_m. 3) Loop headers vary with respect to each other: • Modify one of the headers.	1) $\nexists$ tight loops (L_2, L_1): • Delete L_1 or L_2. • Move a loop (e.g., by INX). • Move a statement between the two loops (e.g., by ICM). • Add/Copy a statement between the two loops (e.g., by FUS) 2) the loop expression changed: • Modify(loop_exp(L_1 or L_2)) (e.g., by LUR, INX, or SMI)

Table 2. Disabling conditions of safety and reversibility.

Safety-disabling actions that possibly disable the safety condition of a transformation are also given in Table 2 for each disabling condition. The following explanation describes how the safety-disabling conditions/ actions for DCE that are given in Table 2 are determined. The disabling condition to the pre-condition "$\exists$ S_i," "$\nexists$ S_i," is ignored since the deletion of S_i does not affect the optimized code. The disabling condition, "$\exists$ $S_l \ni (S_i \delta S_l)$," is obtained by negating the pre-condition "$\nexists$ $S_l \ni (S_i \delta S_l)$." DCE could be disabled by the insertion of a use S_l. The insertion of S_l can be accomplished by adding a new statement, by modifying an existing statement, or by moving a statement onto a path. Since a legal optimization that preserves the semantics of the original program cannot interfere or sever definition-use chains or alter the order in which data is input or output by I/O devices [12], actions that violate the rule for legal transformations are due to changes to the program code by edits and are denoted by † in Table 2. For the DCE example, the movement of S_l on the path so that S_i reaches is due to edits but not the reversal of a legal transformation since the application of a transformation by moving S_l off the path would never occur (it would sever the def-use chain).

(2) Reversibility: In our approach to performing and removing a transformation by primitive actions, a transformation, t_i, is **reversible** if the inverse actions of t_i can be immediately performed. The stored history information, post_pattern, is used to determine whether the inverse actions can be performed. If the post_pattern of a transformation, t_i, is invalidated, there exist subsequent transformations of t_i that changed t_i's post_pattern and made it irreversible. These transformations are affecting transformations that disables the reversibility conditions of t_i. Therefore, given a sequence of transformations, $T = \{t_1, t_2, ..., t_n\}$, the reversibility conditions of transformation t_i can be disabled by its posterior transformations, t_j, where $i < j <= n$. Primitive actions that disable the conditions of reversibility are identified in Table 2. The following explanation describes how the reversibility-disabling actions for DCE are determined. The primitive action for the application of DCE is Delete(S_i) and its inverse action is Add(S_i, orig_loc). If the post-pattern is validated, the inverse action of DCE can be correctly performed. We know the deleted statement S_i can be recovered since it is saved for restoration. If the original location can not be determined, there must be some actions caused by affecting transformations that make the original location of S_i undeterminable. For example, if the context of the original location is deleted or copied by subsequent transformations of DCE, the inverse action, Add(S_i, orig_loc) can not be correctly performed. Therefore, the affecting transformations should be reversed first in order to make DCE reversible.

3.3. Undo Algorithm

This section presents an undo algorithm by which the undo command can be issued to any transformation and only invalidating and invalidated transformations need to be undone. When undoing a transformation, affecting transformations that disable the reversibility of the transformation are reversed first, followed by the reversal of the transformation. Then, the affected transformations are reversed. The interactions of transformations are used as a heuristic to reduce the redundant analysis.

Figure 3 shows an algorithm for undoing transformations in an independent order. The first step in the algorithm (lines 4-14) is to detect and reverse affecting transformations. If the removed transformation is the last transformation applied (t_n), t_n is undone by applying its primitive actions (line 15). If the removed transformation is in any order, the post_pattern of transformation t_i is examined to see whether it is invalidated. If the post_pattern of t_i is invalidated, there must exist some transformations after t_i that change the post_pattern of t_i

```
1  Procedure UNDO(ti)
2  /* undo ti= { ai1, ai2, ..., ain}, where aij are primitive actions, from
     T = { t1, t2, ..., tn }, a sequence of applied transformations */
3  BEGIN
4   if(i<n)
5   THEN BEGIN
6     while(post_pattern(ti) is invalidated)
7      THEN BEGIN
8       /* Undoing affecting transformations */
9       Determine a disabling condition of reversibility for ti;
10      Determine a primitive action that causes the condition;
11      Determine the transformation, tj, that causes the action;
12      UNDO(tj);
13     END {WHILE}
14   END {THEN}
15   Perform inverse actions of ti;
16   Dependence_and_data_flow_update;
17   /* Undoing affected transformations */
18   if(i<n)
19   THEN BEGIN
20    Determine affected regions;
21    For any transformation tk in the affected region
22     DO BEGIN
23     IF (k>i)/*only subsequent transformations may be affected*/
24      THEN BEGIN
25       IF ([ti,tk] is marked "x" in the reverse-destroy table)
26       THEN BEGIN
27        Determine safety conditions of tk, given the events of
            inverse actions of ti;
28        IF !safety(tk)
29         THEN BEGIN
30          UNDO(tk);
31         END {THEN}
32       END {THEN}
33      END {THEN}
34     END {DO}
35   END {THEN}
36 END
```

Figure 3. An undo transformation algorithm.

and create a condition (as listed in Table 2) that disables the reversibility of t_i (line 9). Annotations of applied actions on the program representation are used to determine which actions cause the condition (line 10). The order stamp associated with each primitive action is used to determine the applied transformation (line 11) and the affecting transformation is then reversed to make t_i reversible (line 12). After the reversal of affecting transformations, transformation t_i is reversed by performing its inverse primitive actions (line 15). Next, data flow and data dependence analyses are performed (line 16). Then, the affected transformations are detected and reversed (lines 18-35). Transformations in the affected region due to code changes, data flow changes, and data/control dependence changes are considered as possibly affected transformations (line 20). Line 23 shows that only transformations after t_i (k>i) may be affected. The interactions of transformations are used as a heuristic to reduce the redundant analysis. For each entry "x" of the reverse-destroy row in Table 1, detection of those conditions that cause rippling effects is included in line 25. The disabling conditions of safety are checked for the

determination of affected transformations (line 28). If t_k is not safe due to the removal of t_i, t_k must also be undone (line 30).

4. Analysis of the Undo Transformation Algorithm

In this section, the correctness of the undo transformation algorithm is considered. In order to show that the undo algorithm in Figure 3 is correct, we show that the functional equivalence between the code before and after the removal of transformations is preserved.

I. Claim: Functional equivalence between the code before and after the removal of transformations is preserved.
Proof: Assume that functional equivalence is not preserved. This implies that a transformation has become unsafe and should be reversed, but the undo transformation algorithm does not detect the change in safety. A transformation can become unsafe due to transformation reversal. Thus, it is possible that a transformation becomes unsafe, then safe again, and subsequently unsafe during the removal of a transformation in an independent order. Consider the last time that the transformation becomes unsafe during the rippling of transformation reversal.

Assume that the analysis for affected transformations (line 28 in the algorithm) incorrectly finds the transformation to be safe, so the transformation remains unsafely in the final code. This implies one of the following: (1) The transformation is safe at this point and remains safe until the end of the undo process. This is a contradiction. (2) The transformation is safe during this safety analysis and a later transformation removal causes the transformation to become unsafe. This is a contradiction to the assumption that this is the last time that the transformation becomes unsafe. (3) The transformation is not found to be unsafe because it was already reversed and no longer exists. This implies that an earlier transformation removal caused it to become unsafe, detected its unsafety and reversed it. This is a contradiction to the unsafe transformation still being in effect at the end of the algorithm.

Thus, the analysis for the unsafe transformation must find the transformation to be unsafe and reverse it. This is a contradiction to the assumption that functional equivalence is not preserved. □

5. Implementation and Experiments

This section describes the implementation of transformation application and undo facilities and discusses the experimental design and results of the experimentation. Our primitive functions use most of the Standard Edit Functions (SEF) for the PDG provided by a PDG C Compiler (PDGCC) [5]. The transformation application and undo facilities are implemented in our program parallelization and visualization tool, PIVOT [7]. The PIVOT user interface is implemented on Sun4, SPARC machines under the X Window environment. The X window tools and libraries used for the implementation of PIVOT prototype include TAE+ (a user interface builder), Tcl/tk, and the Open Software Foundation's Motif Toolkit.

The experimentation examined programs in the HOMPACK test suite and pdgccbenches for applicable transformations and enabled transformations. Source code was obtained for HOMPACK from netlib@ornl.gov. HOMPACK is a suite of FORTRAN 77 subroutines for solving non-linear systems of equations by the homotopy methods. The C language is used for the front end of the PDGCC system. Therefore, the FORTRAN programs from HOMPACK were translated into C by using a program, f2c, obtained from netlib@att.com that converts Fortran 77 into C. Another set of test programs are ftp from ftp.edel.edu:/pub/people/pollock. This directory contains some programs that are accepted by pdgcc. The directory contains programs from the Livermore loops, the Stanford benches and some other C programs. Some modifications have been made to the C code so that it could be accepted by pdgcc.

Similar to the experimental studies performed in [13], seven transformations listed in Table 3 were chosen to be a representation sample. Table 3 displays the names and sizes of the programs involved in the experimentation. The table shows the comparison of program sizes using C and the PDG. The number of loops existing in each program is also shown in the table. Numbers in other columns represent number of applicable transformations. Enabled transformations are denoted with comments of enabling transformations.

Some of the interesting results obtained for the HOMPACK test suite are discussed below. (1) Constant propagation (CTP) is applicable for four programs in Table 3. After we examined the applicable points in these programs, we found they do have constant definitions like "one = (float)1.0" that can be propagated to uses of that constant. (2) Copy propagation (CPP) is a transformation that can be frequently applied in the programs. Similar to FUS, an additional statement of the final value of a for loop statement is introduced. This implies that after the application of CPP, some transformations like FUS or INX may be enabled. As shown in Table 3, those programs with a comment (cpp) in the columns of INX and FUS are applicable for transformations INX and FUS after the application of CPP. (3) Loop Interchanging

Source	#stmts	#nodes	#loops	CTP	CPP	DCE	ICM	INX	FUS	LUR
dcpose	73	125	7		6		1			
fode	94	154	12		11		1		1(cpp)	
gfunp	32	57	5		4			1(cpp)		1
gmfads	123	204	10		7		1(cpp)			
hfun1p	67	80	2		2		1(cpp)			
initp	167	226	14		10	2	1			
otputp	34	64	6		4		1			2
polsys	114	161	7		4		1(cpp)			3
polyp	166	260	60		12	2	1	4(cpp) 1(icm)		7
qrfaqf	61	85	4	1	3	1	3			
rhojac	31	40	2		2					
rootnf	84	138	8	1	7	1				1
rootns	72	106	3	1	2	1				1
sclgnp	179	295	28		24		1	2(cpp) 1(icm)		1
sintrp	89	144	8		7					
stepns	189	252	5		3		1			1
tangqf	76	96	5	3	5		1			
bubble	55	148	1	1						
clinpack	163	591	11	1				2	2(inx)	
hanoi	59	147		3						
heapsort	93	247	3	4						
intmm	50	142	5	14				2		5(ctp)
loop1	33	81	2	3						2
loop7	35	117	3	2					1(lur)	3
loop8	48	282	5	12				3		5
loop10	43	159	3					1		3
nsieve	111	281	6	4		2				1
perm	47	118	3							2
puzzle	275	1186	44					28		26
queens	66	216	1							1

Table 3. Applicable transformations in the HOMPACK and pdgccbenches programs.

(INX) is applicable for three programs, gfunp, polyp, and sclgnp. Also due to the introduction of a separate statement by f2c for the definition of the loop final value, two tightly nested loops are not tightly nested anymore after the insertion of a statement between them. Therefore, INX may be enabled by applying CPP (e.g., program sclgnp). (4) Loop Unrolling (LUR) is applicable for eight programs in Table 3. These programs applicable for LUR have constant loop final values. For other programs, LUR may be enabled after the application of constant propagation to the loop initial or the loop final value.

Some of the interesting results obtained for the pdgccbenches are discussed below. (1) CTP is applicable for 9 programs in Table 3. For program intmm, CTP enables five FUS transformations. (2) INX is applicable for five programs. For program clinpack, two FUS transformations can be enabled by applying INX. (3) Three FUS transformations are applicable in two programs, clinpack and loop7 and these transformations are enabled by transformations such as INX and LUR. (4) LUR is applicable for 9 programs in Table 3. No data dependences need to be verified for LUR and LUR will copy the body of a loop that is executed at least once.

6. Discussions

This paper describes a unifying framework to undo code transformations. The technique employs inverse primitive actions and the program dependence graph as the intermediate representation, making it transformation and language independent. Pre and post conditions of transformations are utilized to determine whether an applied transformation remains safe and whether it is immediately reversible. Affecting transformations that disable the reversibility of the transformation are reversed first. Then, the affected transformations determined by checking the safety conditions of transformations are reversed. With the undo facility supported, the user can try different alternatives and undo unpromising transformations.

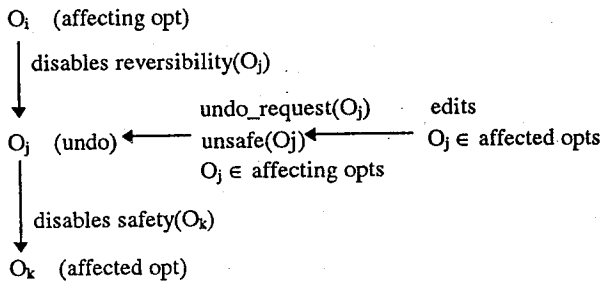

Figure 4. Transformations to be reversed.

When a program is modified by edits, the safety conditions of a transformation can be altered such that the transformation is no longer valid without possibly affecting the program semantics. Since all actions in Table 2 may result from changes to the program code by edits, the reversal of a transformation can be viewed as restricted forms of edits. If a transformation is not safe due to edits, it is then reversed by using our undo algorithm.

Figure 4 shows the summary of transformations to be reversed in an independent order. A transformation, O_j, needs to be reversed if it is under the user's undo request, if it is unsafe due to the modification by edits or the transformation rippling (affected by reversing preceding transformations), or if it is an affecting transformation that disables the reversibility of another transformation. Figure 4 also shows the sequence of reversing transformations. Affecting transformations are reversed first, then the transformation to be undone, and then affected transformations due to the transformation rippling (reverse-destroy).

The results in this work are encouraging and this research will continue. Some preliminary results for incremental transformations have been developed, and further investigation is needed for various primitive edit functions. The next step in this research will be to develop techniques for incrementally parallelizing programs by edits. Experiments should be conducted to obtain performance results for a sequence of transformations in different application points after the full implementation of data dependence update. Also, experiments could be conducted to compare the results of undoing transformations in any order with the effort required for total recompilation and reanalysis of transformations when starting all over again. Another step will be to investigate techniques to automatically generate code for the detection of the disabling actions of the safety and reversibility conditions of transformations from the transformation specifications.

References

1. A. Aho, R. Sethi, and J. Ullman, in *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA, 1986.

2. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 617-640, 1988.

3. F. E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations," in *Design and Optimization of Compilers, Edited by Randall Rustin*, pp. 1-30, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, March 1971.

4. R. Allen and K. Kennedy, "Automatic Loop Interchange," in *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pp. 233-246, 1984.

5. D. Berson, R. Bodik, C. N. Fiechter, and P. A. Kamp, "PDG C Compiler User Manuals," Department of Computer Science, University of Pittsburgh, 1993.

6. C. R. Dow, M. L. Soffa, and S. K. Chang, "An Efficient Technique to Undo Code Transformations," in *Proceedings of 1994 International Conference on Parallel and Distributed Systems*, pp. 392-397, Hsinchu, Taiwan, December, 1994.

7. C. R. Dow, S. K. Chang, and M. L. Soffa, "PIVOT: A Program Parallelization and Visualization Environment," in *1994 International Computer Symposium*, pp. 671-676, Hsinchu, Taiwan, December 1994.

8. K. Kennedy, K. McKinley, and C.-W. Tseng, "Interactive Parallel Programming Using the ParaScope Editor," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 329-341, July 1991.

9. D. B. Loveman, "Program Improvement by Source-to-Source Transformation," *JACM*, vol. 24, no. 1, pp. 121-145, January 1977.

10. L. L. Pollock and M. L. Soffa, "Incremental Global Reoptimization of Programs," *ACM Trans. on Programming Languages and Systems*, vol. 14, no. 2, pp. 173-200, April 1992.

11. C. D. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," in *Proceedings of 1989 International Conference on Parallel Processing*, pp. 39-48, St. Charles, Illinois, 1989.

12. D. Whitfield and M. L. Soffa, "An Approach to Ordering Optimizing Transformations," in *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, pp. 137-146, March 1990.

13. D. Whitfield, "A Unifying Framework for Optimizing Transformations," Ph.D. Thesis, University of Pittsburgh, Technical Report 91-24, 1991.

14. M. Wolfe, "Software Optimization for Supercomputers," *Supercomputers, Class VI Systems, Hareware and Software, Edited by S. Fernbach*, pp. 221-238, Elsevier Science Publishers B. V. (North-Holland), 1986.