Undoing Code Transformations in an Independent Order†

Chyi-Ren Dow dow@cs.pitt.edu
Mary Lou Soffa soffa@cs.pitt.edu
Shi-Kuo Chang chang@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

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 technique to remove such transformations. 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. Transformations that affect the immediate removal of a transformation must be identified and removed. Other transformations made invalid by the removal of a transformation must also be undone. The technique employs inverse primitive actions, making the technique transformation independent. The enabling and disabling interactions of transformations are used to drive the process, thereby reducing redundant analysis when undoing transformations.*

1. Introduction

Parallel architectures use various forms of parallelism for increased computational power. Programmers, eager to speed up sequential code on parallel machines, depend on parallelizing compilers to enhance or expose parallelism in sequential programs and to generate highly parallelized code. Code transformations are important components in parallelizing compilers and are used to restructure code to enable the exploitation of parallelism in programs.

Applying a transformation does not always guarantee a time or space benefit. Also, the application of a transformation may invalidate conditions for another transformation which may be more beneficial [20]. Due to the interaction between schedulers and transformations, the application of a transformation may produce a worst schedule than before the transformation was applied [19]. Because it is not clear whether or not a transformation will be effective when it is applied, it may be necessary to remove it if it is not beneficial to parallelism.

† Partially supported by NSF under Grants CCR-9109089 and IRI-9002180 to the University of Pittsburgh

An undo feature has been presented [5] that allows users to undo code transformations in the reverse application order. The user may also want to undo transformations in an independent order in order to remove inappropriate transformations while maintaining beneficial ones. 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. After a transformation has been removed, disabled transformations would have to be removed also. When a program is modified by edits, the safety conditions of a transformation can be altered such that the transformation is no longer applicable without possibly affecting the program semantics. This kind of transformation is defined to be unsafe and needs to be removed. However, all other transformations may be unaffected and should remain in the code. In order to reduce the redundant analysis that is performed for redoing all transformations in response to program edits, only unsafe transformations should be identified and removed. Thus, techniques for undoing transformations in an order independent of application order become valuable when undoing transformations at the user's directive or when a program is modified by edits.

In this paper, we present a technique for undoing code transformations in an order independent of application order. The technique employs inverse primitive actions to undo transformations, making it transformation independent. A two-level program representation is developed that allows the application of both traditional optimizations and parallelizing transformations. It also supports the reversal of transformations by maintaining information about applied transformations. Pre and post conditions of transformations are utilized to determine whether an applied transformation remains safe and whether it is immediately reversible. The remainder of this paper is organized as follows. In Section 2, we discuss schemes developed to reverse traditional optimizations and to undo code transformations in reverse order. Section 3 describes a two-level program representation

used in our scheme. The reversing transformation scheme is presented in Section 4, followed by an algorithm and an example for undoing transformations in Section 5. Finally, conclusions are drawn in Section 6.

2. Background

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 [13, 20]. 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 occur in that removing a transformation destroys the safety of other transformations that must also be removed.

A definition of *Undo* might be to restore a user's program or application to a previous state [10]. Recovery features to do this have been built in many systems such as editors [23] and database systems [17]. Although numerous parallelizing compilers and parallelization systems, such as PTRAN [4], ParaScope Editor [8], and Parafrase II [14] have been designed and implemented, 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 [13]. It works on intermediate-level program representations, namely a control flow graph and dag representation, with history information of optimizations placed on dag nodes. These annotations represent code that has been eliminated, relocated, or replaced by optimizations. Algorithms are given for determining which optimizations are no longer safe after a program change. The representation is incrementally updated to reflect the current optimizations in the program. The annotations to the intermediate representation are dependent on the particular transformation applied. When undoing transformations, actions are guided by the enabling and disabling interactions of transformations. 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.

Action	Inverse Action
Delete (a)	**Add** (orig_location, -, a)
Copy (a, location, c)	**Delete** (c)
Move (a, location)	**Move** (a, orig_location)
Add (location, description, a)	**Delete** (a)
Modify (exp(a), new_exp)	**Modify** (new_exp(a), exp)

Table 1. Actions and inverse actions.

Approaches to performing transformations and undoing transformations in reverse order of application by using primitive actions have been presented in [5, 20]. For undo in order, the first time the undo command is issued, the last transformation is undone. Consecutive repetitions of the undo command continue to reverse earlier transformations. Each transformation is undone by applying its inverse actions. The five primitive actions used in the scheme as well as their inverse actions are listed in Table 1. With appropriate transformation history maintained (e.g., the original locations of moved and deleted statements), the reversal of transformations in order can be performed immediately by the corresponding inverse actions. Due to the complexity of transformation interactions (transformation enabling and disabling), the reversal of transformations in an independent order may not be applicable or safe by directly performing the inverse actions of the transformations.

3. Program Representations

A program representation is used to display the relevant information needed for analyzing and generating the code for the program. Various program representations have been used in optimizing and parallelizing compilers, such as the Abstract Syntax Tree (AST), the Control Flow Graph (CFG) [1], the Data Dependence Graph (DDG) [9], the Program Dependence Graph (PDG) [7], and the Dependence Flow Graph (DFG) [11]. Different representations may have different effects on the design, analysis and implementation of code transformations. Traditional optimizations typically work on low-level program representations such as the dag representation of basic blocks. A dag for an expression represents the data dependences in the expression. The statements in a basic block can be represented by a dag to show how the value computed at one statement is used in subsequent state-

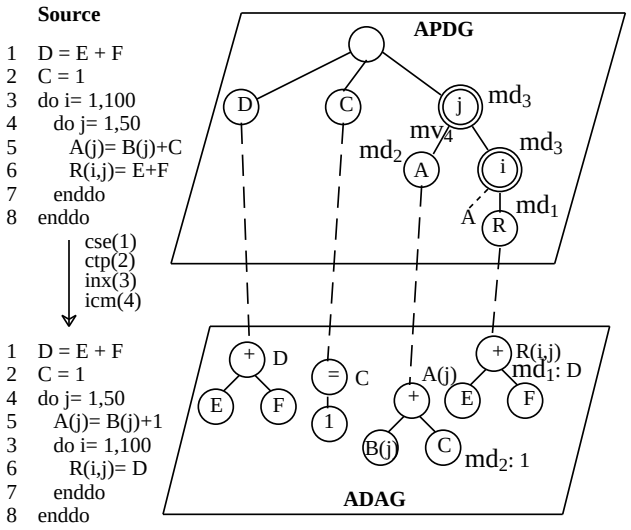

Figure 1. A two-level program representation.

ments. Parallelizing transformations typically work on high-level program representations such as the PDG. The PDG is recognized as a useful representation in parallelizing techniques for a vector or parallel machine. It is a program representation that makes explicit both the data and control dependences in a program and explicitly shows what statements can be executed in parallel. The nodes in the PDG are statements, predicate expressions, and region nodes. The edges in the PDG represent control dependences and various data dependences.

Parallelizing compilers apply both parallelizing (high-level) transformations and traditional (low-level) optimizations to exploit parallelism in sequential programs. Many of these compilers use two different program representations, a high level one for parallelization and a low level one for scalar optimizations. Without integrating the high-level and low-level representations, the compilation requires two stages for parallelization and scalar optimization and these two phases can not be freely intermixed. Due to the interactions between scalar optimization and parallelization, integrating scalar and parallelizing transformations can improve the use of parallelism and memory hierarchy [16]. Our two-level representation integrates two program representations, a high level one - PDG and a low-level one - DAG, to allow the application of both parallelizing transformations and traditional optimizations. Advantages of using this model include

(1) Optimizing and parallelizing transformations can be freely intermixed.

(2) Transformations can use both high/low level information (e.g., data dependence/ data flow).

(3) Code generation and code scheduling for architectures with different granularity can be supported by the two-level representation.

(4) With appropriate information annotated on the representation (see Section 4.1), an undo facility for both optimizing and parallelizing transformations can be supported.

Figure 1 shows an example of two-level program representation for a program segment that is restructured by the common subexpression elimination (cse), constant propagation (ctp), loop interchanging (inx), and invariant code motion (icm) optimizations. The common subexpression evaluation E+F is replaced by the value of D at labeled statement 6. The reference to variable C at statement 5 is replaced by the constant 1. The tightly nested loops (labeled statements 3 and 4) are interchanged and the invariant statement 5 is moved out of loop at statement 3. In Figure 1, annotations on the program representation, such as md_3, are used for reversing transformations and are discussed in the next section.

4. Reversing Transformation Scheme

A scheme for reversing transformations using the two-level program representation is discussed in this section. First of all, information to be stored for the reversal of transformations is presented. Secondly, pre and post conditions of transformations used to determine whether a transformation remains safe and whether it is immediately reversible are presented. Thirdly, interactions of code transformations are discussed. Finally, an event driven regional undo approach is discussed.

4.1. Information to be Stored

In order to allow the reversal of transformations, sufficient information must be recorded to keep a history of all existing transformations. Information is required to ensure correct detection and removal of invalidated 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 by transformation independent annotations. A pre_pattern notation is used to determine whether the transformation remains safe and a post_pattern is used to determine whether the transformation is immediately reversible as discussed in the following sections.

Table 2 shows the pre_patterns, primitive actions, and post_patterns for a set of transformations. 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

Transformation	Pre_pattern	Primitive Actions	Post_pattern
Dead Code Elimination (DCE)	Stmt S_i; /*dead code*/	Delete(S_i);	Del_stmt S_i; ptr orig_loc;
Constant Propagation (CTP)	Stmt S_i: type(opr_2) == const; Stmt S_j: opr(pos) == S_i.opr_2;	Modify(opr(S_j,pos), S_i.opr_2);	Stmt S_i: opr(pos)== S_i.opr_2;
Common Subexpression Elimination (CSE)	Stmt S_i: A= B op C; Stmt S_j: D= B op C;	Modify(exp(S_j,B op C), A);	Stmt S_j: D= A;
Invariant Code Motion (ICM)	Loop L_1; Stmt S_i;	Move(S_i,L_1.prev);	Stmt S_i; ptr orig_location;
Loop Interchanging (INX)	Tight Loops (L_1, L_2);	Copy(L_1, L_{tmp}); Modify(L_1, L_2); Modify(L_2, L_{tmp});	Tight Loops (L_2, L_1);

Table 2. Information to be stored.

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 [5, 21]. 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.

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. The two level program representation becomes an Augmented DAG (ADAG) and an Augmented PDG (APDG). 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 as discussed in Section 4.2. Figure 2 shows the transformation annotations for the transformations applied in which md, mv, and del are abbreviation of modify, move, and delete.

As shown in Figure 1, information of the four applied transformations is retained and annotated on the program representation. In the high level representation APDG, modified and moved code elements are annotated with its corresponding history of applied primitive actions. In the low level representation ADAG, a global common subexpression elimination (statement 6: E+F) is represented by the original subexpression tree with the root annotated with modified variable D. A constant propagated operand in statement 5 is represented by its original operand with the annotation of the propagated constant value.

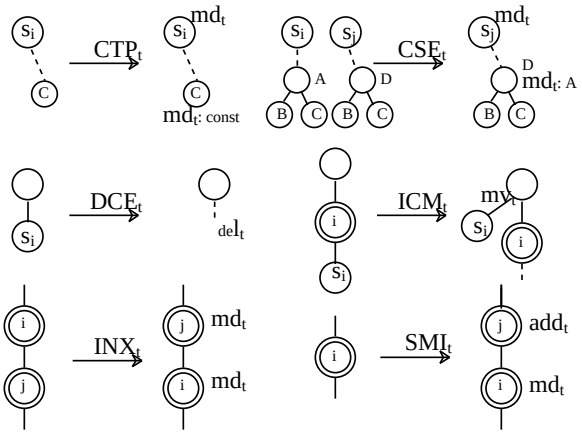

Figure 2. Annotations based on primitive actions.

4.2. 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. Post conditions are conditions that result from applying the transformation. The existence of post-conditions determines whether the transformation remains safe. Post conditions of a transformation can be altered by applying and removing transformations. Two kinds of post conditions of transformations 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	Disabling Conditions of Safety	Disabling Conditions of Reversibility
Dead Code Elimination (DCE) *Pre-conditions*: $\exists$ $S_i \wedge \nexists$ $S_l \ni (S_i \delta_{=} S_l)$.	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)

Table 3. Disabling Conditions of Safety and Reversibility.

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

Table 4. Perform-create (reverse-destroy) interactions.

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 [13]. 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 $1 <= j < i$. Table 3 gives pre-conditions, safety-disabling conditions, and reversibility-disabling conditions for DCE. Conditions for other transformations are described in [6]. The safety-disabling conditions of a transformation are determined by negating the pre-condition of a transformation. The pre-condition information can be obtained automatically if the specification approach using primitive transformations is taken in the development of transformations.

Safety-disabling actions that possibly disable the post-condition of a transformation are also given in Table 3 for each disabling condition. The following explanation describes how the safety-disabling conditions/ actions for DCE that are given in Table 3 are determined. The disabling condition to the pre-conditions "$\exists$ 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 [20], actions that violate the rule for legal transformations are due to changes to the program code by edits and are denoted by † in Table 3. 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 called affecting transformations to 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 3. 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). The post_pattern of DCE in Table 2 consists of finding the deleted statement S_i and its original location. 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.

4.3. Interactions of Code Transformations

The application of one transformation may enable or disable other transformations [13, 20]. Table 4 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. Interactions of code transformations may occur by one transformation enabling the application of another transformation that previously could not be applied. Thus, enabling interactions are perform-create dependencies. Since 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 [13]. 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 are considered as possibly affected transformations.

4.4. An Event Driven Regional Undo

When a transformation is to be removed, subsequently applied transformations need to be checked to see whether they are affecting, affected, or unrelated (unaffected/ unaffecting) transformations. One approach is to examine all the following transformations but this may be too time consuming due to the redundant analysis of unrelated transformations if the number of transformations is large. Our approach is to employ an event driven regional undo technique to detect transformations only in affected regions. Thus, our approach is based not only on the order coordinate (transformation ordering) but also on the space coordinate (affected regions).

An affected region is defined as the region of a program with code changes (e.g., code reordering or modification) or data flow or data/ control dependence changes. In our two level program representation, an affected region in the low level representation is defined as a basic block with expression or data flow changes and an affected region in the high level representation is

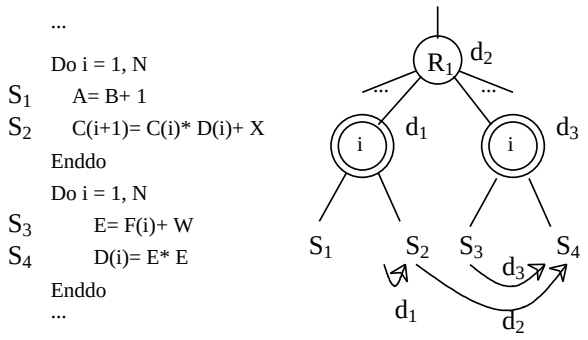

Figure 3. Summary of data dependences on region nodes.

defined as a region node as well as nodes dependent on it with code, data dependence, or control dependence changes. Various techniques for data dependence summaries such as the hierarchical dependence graphs [18] and the intrablock and interblock data dependences [2] can be used to improve the data dependence verification process. In order to ease the determination of affected regions on the PDG for parallelizing transformations, data dependence summaries are annotated on the region nodes. We define the least common region node, denoted by LCR(s_i, s_j), as a region node and the least common control ancestor of nodes s_i and s_j in the control dependence tree of the PDG [3, 7]. Each data dependence on the least common region node of the source and sink of that dependence is annotated. As shown in Figure 3, each data dependence is summarized on the least common region node of the source and the sink of the dependence. The summary of data dependences on region nodes can be used not only to determine affected regions with dependence changes but also to determine applicable transformations without visiting all nodes under the region nodes. For example, it can be determined whether the two loops in Figure 3 can be fused by checking only the inter-region data dependence (i.e. d_2) on R_1. If d_2 is not a loop fusion (FUS) prevented dependence, FUS can be applied directly without visiting all nodes under the two loops. Data flow and data dependence updates can be performed incrementally using various techniques including incremental data flow analysis using the CFG [1, 12], the DDG [22], and the PDG [7] and the incremental data dependence analysis [15].

5. Undo Algorithm and Example

For undo in an independent order, the undo command can be issued to any transformation and only invalidating and invalidated transformations need to be undone. This section presents an algorithm and an example of undoing transformations in an independent order.

```
 1  Procedure UNDO(t_i)
 2  /* undo a transformation t_i from a sequence of applied
       transformations, T = { t_1, t_2, ..., t_n } */
 3  BEGIN
 4    while(post_pattern(t_i) is invalidated)
 5      THEN BEGIN
 6        /* Undoing affecting transformations */
 7        Determine a disabling condition of reversibility for t_i;
 8        Determine a primitive action that causes the condition;
 9        Determine the transformation, t_j, that causes the action;
10        UNDO(t_j);
11      END {WHILE}
12    Perform inverse actions of t_i;
13    Dependence_and_data_flow_update;
14    /* Undoing affected transformations */
15    Determine affected region due to code, data flow,
      and data dependence changes;
16    For any transformation t_k in the affected region
17      DO BEGIN
18        IF (k > i)          /* only subsequent transformations */
19          THEN BEGIN  /* may be affected                      */
20            IF ([t_i,t_k] is marked "x" in the reverse-destroy table)
21            THEN BEGIN
22              Determine safety conditions of t_k, given the events
                of inverse actions of t_i;
23              IF !safety(t_k)
24                THEN BEGIN
25                  UNDO(t_k);
26                END {THEN}
27            END {THEN}
28          END {THEN}
29      END {DO}
30  END
```

Figure 4. An undo transformation algorithm.

5.1. Undoing Transformation Algorithm

As discussed in previous sections, 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 and an event driven regional undo approach is used to reduce the redundant analysis. Figure 4 shows an algorithm for undoing transformations in an independent order. The first step in the algorithm (lines 4-11) is to detect and reverse affecting transformations. 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 and create a condition (as listed in Table 3) that disables the reversibility of t_i (line 7). Annotations of applied actions on the program representation are used to determine which actions cause the condition (line 8). The order stamp associated with each primitive action is used to determine the applied transformation (line 9) and the affecting transformation is then reversed to make t_i reversible (line 10). After the reversal of affecting transformations, transformation t_i is reversed by performing its inverse primitive actions (line 12). Next, data flow and data dependence analyses are performed (line 13). Then, the affected transformations are detected and reversed (lines 15-29). Transformations in the affected region due to code changes, data flow changes, and data/control dependence changes are considered as possibly affected transformations (line 15). Line 18 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 4, detection of those conditions that cause rippling effects is included in line 20. The disabling conditions of safety are checked for the determination of affected transformations (line 22). If t_k is not safe due to the removal of t_i, t_k must also be undone (line 25).

5.2. An Example

Figure 1 is an example of a program segment restructured by four transformations in the order of CSE, CTP, INX, and ICM. Since the post_patterns of CSE and CTP exist and the original common subexpression, E+F, and the original constant use, C, are retained on the representation, CSE and CTP can be reversed immediately by deleting their annotations on the program representation [13]. Also, the reversal of ICM can be immediately applied by performing its inverse actions since it is the last transformation applied. To undo INX, the post_pattern of INX, Tight Loops (L_j, L_i), is invalidated due to the movement of statement 5 in between the tightly nested loops. The action mv_4 that moves statement 5 results from the fourth transformation, ICM. Therefore, both transformations must be undone with undoing ICM first in order to undo the loop interchanging.

6. Conclusions

This paper describes a technique to undo transformations in an order independent of application order. An integrated high-level and low-level representation is presented that allows the application of both traditional optimizations and parallelizing transformations. 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.

The technique of undoing code transformations in an independent order is currently being implemented in a visualization system for parallelizing programs [5]. With the undo facility supported in our system, the user can try different alternatives and undo unpromising transformations. The next step in this research will be to perform experimental studies for undoing transformations. 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. D. Bernstein and M. Rodeh, ‘‘Global Instruction Scheduling for Superscalar Machines,’’ in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 241-255, June 1991.
3. R. Cytron, J. Ferrante, and V. Sarkar, ‘‘Experiences Using Control Dependence in PTRAN,’’ in *Languages and Compilers for Parallel Computing*, ed. D. Gelernter, A. Nicolau, and D. Padua, pp. 186-212, The MIT Press, 1990.
4. R. Cytron, J. Ferrante, and V. Sarkar, ‘‘Experiences Using Control Dependences in PTRAN,’’ in *Languages and Compilers for Parallel Computing*, ed. D. Gelernter, A. Nicolau, and D. Padua, The MIT Press, 1990.
5. C. R. Dow, S. K. Chang, and M. L. Soffa, ‘‘A Visualization System for Parallelizing Programs,’’ in *Proceedings of Supercomputing ’92*, pp. 194-203, Minneopolis, MN , November 1992.
6. C. R. Dow, ‘‘PIVOT: A Program Parallelization and Visualization Environment,’’ Ph.D. Thesis, Department of Computer Science, University of Pittsburgh, Technical Report 94-22, June 1994.
7. J. Ferrante, K. J. Ottenstein, and J. D. Warren, ‘‘The Program Dependence Graph and its Use in Optimization,’’ *ACM TOPLAS*, vol. 9, no. 3, pp. 319-349, July 1987.
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. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, ‘‘Dependence Graphs and Compiler Optimizations,’’ in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 207-218, Williamsburgh, VA, January 1981.
10. G. B. Leeman, Jr., ‘‘A Formal Approach to Undo Operations in Programming Languages,’’ *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, pp. 50-87, January 1986.
11. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill, ‘‘Dependence Flow Graphs: An Algebraic Approach to Program Dependencies,’’ in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 67-78, January 1991.
12. L. L. Pollock and M. L. Soffa, ‘‘An Incremental Version of Interactive Data Flow Analysis,’’ *IEEE Trans. on Software Engineering*, vol. 15, no. 11, pp. 1537-1549, December 1989.
13. 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.
14. 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.
15. C. M. Rosene, ‘‘Incremental Dependence Analysis,’’ Rice COMP TR90-112, Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX, March 1990.
16. S. W. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy, ‘‘Integrating Scalar Optimization and Parallelization,’’ in *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pp. 137-151, Santa Clara, CA, August 1991.
17. J. S. M. Verhofstad, ‘‘Recovery Techniques for Database Systems,’’ *ACM Comput. Surv.*, vol. 10, no. 2, pp. 167-196, June 1978.
18. J. Warren, ‘‘A Hierarchical Basis for Reordering Transformations,’’ *Conference Record of 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 272-282, ACM, New York, Salt Lake City, UT, January 1984.
19. T. Watts, R. Gupta, and M. L. Soffa, ‘‘Techniques for Integrating Parallelizing Transformations and Compiler Based Scheduling Methods,’’ in *Proceedings of Supercomputing ’92*, pp. 830-839, Minneapolis, MI, November 1992.
20. 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.
21. D. Whitfield and M. L. Soffa, ‘‘Automatic Generation of Global Optimizations,’’ in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 120-129, June 1991.
22. M. Wolfe and U. Banerjee, ‘‘Data Dependence and its Application to Parallel Processing,’’ *International Journal of Parallel Programming*, vol. 16, no. 2, pp. 137-178, 1987.
23. Y. Yang, ‘‘Anatomy of the design of an undo support facility,’’ *Int. J. Man-Machine Studies*, vol. 36, pp. 81-95, 1992.