I/O Performance Improvement Through the Use of Code Transformations

Wayne Chen Tu and Ernst L. Leiss
Department of Computer Science, University of Houston, Houston, Texas 77204-3475, USA
coscel@cs.uh.edu

Abstract

Virtual memory frees the programmer from the time consuming burden of ensuring that the algorithms used operate efficiently on the data set. However, a carelessly designed program may perform unnecessary page transfers between main memory and secondary memory. By applying code transformations we can utilize the data set residing in main memory more efficiently and reduce the need for frequent page transfers. Our main objective is to restructure programs by using loop distribution and loop interchange. We present an algorithm that determines which code transformation technique, or combination of techniques, should be applied in order to reduce the memory requirements of loops. Loop distribution and loop interchange are not always semantically valid because of data dependences, but other transformation techniques can be used in order to eliminate those dependences. By applying these techniques we can reduce the amount of memory that is required for an iteration of a loop and we can adjust the access pattern to match the storage pattern more closely, thus reducing unnecessary page transfers and improving program performance.

1. INTRODUCTION

Virtual memory has made programming easier by relieving the programmer from the burden of storage management. However, a carelessly designed program can cause virtual memory to degrade performance by performing unnecessary page transfers. This usually occurs when the access pattern does not match the storage pattern and so a page of data that is read into main memory from secondary memory is not fully utilized before it is paged out. A classic example of this is the following FORTRAN code fragment:

```
Do 10 I = 1, 1024
    Do 10 J = 1, 1024
        A(I,J) = I + J
10  Continue
```

The matrix A is stored by columns (default for FORTRAN) but the code accesses A by rows. If the page size is 1024 words, main memory holds 1000 pages, and the replacement algorithm of virtual memory is LRU (Least Recently Used), 1,048,576 page transfers are done. If we transform the code so that it accesses the data by columns instead of rows, the transformed code requires only 1024 page transfers!

The goal of code transformation is to find a technique that will either reduce the amount of memory that is required or modify the access pattern so that it follows the storage pattern more closely. We can reduce the amount of memory that is required by modifying loops. Consider the following code:

```
Array A, B, C, D 1024
Do 10 I = 1, 1024
    A(I) = I + 2*I; B(I) = I + 3*I; C(I) = 4/(3*I + 2); D(I) = I
10  Continue
```

The loop requires all four arrays for each iteration. Assuming that each array is contained in a single page of memory, the loop requires four pages of memory. There are two ways to reduce the amount of memory that is required. The first is to break the loop into smaller pieces:

```
Array A, B, C, D 1024
Do 10 I = 1, 1024
    A(I) = I + 2*I
10  Continue
Do 20 I = 1, 1024
    B(I) = I + 3*I
20  Continue
Do 30 I = 1, 1024
    C(I) = 4/(3*I + 2)
30  Continue
```

```
Do 40 I = 1, 1024
    D(I) = I
40 Continue
```

Then each loop will only require one page of memory. The second method is to divide the range of the loop variable so that less of each array is needed.

```
Array A, B, C, D 1024
Do 10 I = 1, 256
    A(I) = I + 2*I; B(I) = I + 3*I; C(I) = 4/(3*I + 2); D(I) = I
10 Continue
Do 20 I = 257, 512
    A(I) = I + 2*I; B(I) = I + 3*I; C(I) = 4/(3*I + 2); D(I) = I
20 Continue
Do 30 I = 513, 768
    A(I) = I + 2*I; B(I) = I + 3*I; C(I) = 4/(3*I + 2); D(I) = I
30 Continue
Do 40 I = 769, 1024
    A(I) = I + 2*I; B(I) = I + 3*I; C(I) = 4/(3*I + 2); D(I) = I
40 Continue
```

Each loop has reduced the range of access so that only a fourth of the arrays is needed. Therefore theoretically each loop should only need one page of memory. This method works only if the granularity of memory fits the range of the loop: if a page of memory holds 256 words and virtual memory can hold a total of 1024 words, then each of the loops will only read into memory the page that contains the elements that it needs, thus requiring a total of only 16 page transfers. But if a page holds 512 words and virtual memory can hold a total of 1024 words, the loops will require a total of 4096 page transfers. This technique is useful only if the granularity of memory is larger than the size of the range. Therefore splitting up the range of indices of an array may or may not improve the I/O performance.

Chapter 2 discusses dependences and dependence analysis. Chapter 3 reviews the most common types of code transformation techniques and summarizes why they can or cannot be used. Chapter 4 lists the transformation techniques that can be used and introduces the combinations of techniques that will be used in the algorithm. Chapter 5 provides different examples of the algorithm being applied. Chapter 6 summarizes the results, discusses limitations, and suggests future enhancements.

2. DATA DEPENDENCES

Loops are of particular interest in optimizing code because most of the execution time occurs in loops. So if the I/O behavior of a loop can be improved through the use of code transforming techniques without affecting the I/O behavior of the remaining code, then the entire program can be improved. Not all transformation techniques are semantically valid because of dependences within the loop. There is an extensive theory of data dependences and their determination. We refer to [ZC91] and [L95].

3. CODE TRANSFORMATION TECHNIQUES

We present various code transformation techniques and examine each to determine if it is useful in improving I/O behavior.

3.1 Loop Distribution

Loop distribution, also known as loop fusion, is used to distribute operations in order to reduce explicit synchronization. Loop distribution will only change the execution order of statements; so it will not eliminate dependences, but it will change loop carried dependences into loop independent dependences.

Loop distribution is a very valuable transformation for improving I/O behavior. It can help reduce I/O in two ways. The first is by reducing the amount of memory required for each loop iteration. If each variable is stored on a single page, the original loop requires six pages for each iteration while its distributed version requires at most three, or 50% less. The second is by breaking up the loops into simpler loops which are more likely candidates for other techniques, e. g., loop interchange. Consider the code:

```
Do 10 I = 1, 1024
    Do 10 J = 1, 1024
        A(I,J) = B(I,J) + C(I,J)
        D(I,J) = A(I-1,J+1) * 2
10 Continue
```


This nested loop cannot be interchanged because of the loop preventing dependence existing between S_1 and S_2, but the statements can be distributed into:

```
Do 10 I = 1, 1024
   Do 10 J = 1, 1024
      A(I,J) = B(I,J) + C(I,J)
10 Continue
Do 20 I = 1, 1024
   Do 20 J = 1, 1024
      D(I,J) = A(I-1,J+1) * 2
20 Continue
```

Now each of the nested loops can be interchanged and the values for D will be correct.

```
Do 10 J = 1, 1024
   Do 10 I = 1, 1024
      A(I,J) = B(I,J) + C(I,J)
10 Continue
Do 20 J = 1, 1024
   Do 20 I = 1, 1024
      D(I,J) = A(I-1,J+1) * 2
20 Continue
```

3.2 Loop Interchange

Loop interchange is used to interchange loop variables so that a loop carried dependence can be moved outwards so that the inner loops can be vectorized ([L95],[ZC91]).

Loop interchange is a very valuable code transformation technique because by interchanging loops we can modify the access pattern to match the storage pattern more closely. The objective is to use each page more fully and to eliminate unnecessary page transfers. Consider the following code:

```
Do 10 I = 1, N
   Do 10 J = 1, N
      Do 10 K = 1, N
         S:  C(I,J) = C(I,J) + A(I,K) * B(K,J) + D(I,K)
10 Continue
```

Let A, B, C and D (of size 1024x1024) be stored in column major order. Assume N = 1024, each page holds 1024 words, and virtual memory can hold 1000 pages in memory. Thus, each column of each array is stored on exactly one page. In the code above, A, C and D are accessed by rows, but matrix B is accessed by columns. To find the value of C(1,1), virtual memory will have to read into memory exactly 2050*1024 pages. Therefore the code requires 2050*1024*1024 page transfers. If we interchanged the I with the K loop, the matrices A, C and D are accessed column-major while only B is accessed row-major. To find the value for C(1,1) now, virtual memory will need to read into memory exactly 1024*1024 + 3 pages. Thus for the modified code, a total of (1024*1024 + 3)*1024 page transfers occurs: by interchanging the loops we get an improvement of almost 50%.

3.3 Alignment

Alignment is used to reduce the need for explicit synchronization between processors on parallel machines. It does this by changing loop carried dependences into loop independent dependences [L95]. For I/O performance improvement, alignment is not useful because it modifies statements to be smarter so that a larger group of statements can be parallelized together. Alignment does not aid in reducing memory requirements nor does it help modify the access pattern to match the storage pattern.

3.4 Replication (Variable Copying)

Replication breaks a dependence cycle between statements, by replicating the variable which causs the cycle into a temporary variable and then using that temporary variable. Consider the following code:

```
Do 10 I = 1, N
   S1:  A(I) = B(I) + C(I);   S2:  D(I) = A(I) + A(I+1)
10 Continue
```

There is a loop cycle created by the true dependence from S_1 to S_2 and the anti dependence from S_2 to S_1. We break the cycle by removing the anti dependence:

```
Do 10 I = 1, N
   S0:  Atemp(I) = A(I+1);   S1:  A(I) = B(I) + C(I);   S2:  D(I) = A(I) + Atemp(I)
10 Continue
```

Replication is a useful transformation technique because it can be used to break dependence cycles so that loop distribution can be applied. The original code above could not be distributed because of the dependence cycle, but the new code with the replicated array can be distributed:

```
Do 10 I = 1, N
   S0:  Atemp(I) = A(I+1)
10 Continue
Do 20 I = 1, N
   S1:  A(I) = B(I) + C(I)
20 Continue
Do 30 I = 1, N
   S2:  D(I) = A(I) + Atemp(I)
30 Continue
```

The original code required all four arrays for each iteration of the loop, but this new code only requires three arrays at the most for an iteration of a loop. Thus replication is very useful because it allows us to distribute loops that ordinarily could not be distributed [L95] [ZC91].

3.5 Node Splitting

Node splitting is a technique that is similar to variable copying. Its main purpose is to break up a node (statement) so that other components of the statement can be taken out and vectorized. Node splitting is a useful technique to use for I/O behavior improvement because it allows us to break up statements so that we can reduce the memory requirements of the loop.

3.6 Multiversion Loops

Multiversion loops are used whenever we are unable to transform code because we don't have enough syntactic information. But we may create two or more versions of the loop so that at run time one of the resulting loops will be executed, which may improve the I/O performance [L95].

3.7 Strip Mining

Strip mining is a technique that is used to take advantage of the target system, such as the vector length of a vector register [L95]. It is not a useful technique for I/O performance improvement because its main purpose is to take advantage of special constructs that exist on vector or parallel machines and these special features will not help to improve I/O behavior.

3.8 Loop Collapsing

Loop collapsing is a technique that creates a single, larger loop out of two smaller loops so that more statements can be vectorized together. It is not a useful technique for I/O behavior improvement because it does not reduce memory requirements [L95].

3.9 Loop Fusion

Loop fusion is a similar technique to loop collapsing except that it operates on separate loops instead of nested loops. It is not a useful technique for I/O performance improvement because it combines loops so that loop overhead is reduced. When two loops are fused together, the memory requirements increase, which is exactly the opposite effect of loop distribution. Therefore loop fusion should not be used [L95].

3.10 Wavefront Method

The wavefront method is a technique that changes the access pattern of a multidimensional iteration space to follow a wavefront rather than a particular index. It will not aid in improving I/O behavior because it accesses the array elements diagonally which is worse than accessing them along their storage pattern. Therefore the wavefront method should not be used [L95].

3.11 Scalar Expansion

Scalar expansion is a transformation that can be applied to a scalar variable within a loop. It creates a copy of the variable for each iteration of the loop nest by replacing the variable with an appropriately dimensioned array. This is used to eliminate dependences involving the variable so that the code may be vectorized. Scalar expansion must be applied very carefully because if the range is too large, then it may cause a large, undesirable increase in a program's memory requirements. If this occurs then we can either refrain from applying the transformation or restrict the application to lie within the innermost loop. In general, this transformation may be used if it will remove dependence cycles and allow statements to be distributed [ZC91].

3.12 Array Shrinking

Array shrinking is the converse transformation of scalar expansion. It is used to undo the effects of scalar expansion. This technique is not used otherwise because determining which arrays can be reduced into a

single scalar variable is very difficult because there are too many possible side effects. Therefore this is not a useful technique for I/O performance improvement since is it only used to reverse the effects of scalar expansion [ZC91] [L85].

3.13 Index Set Splitting

Index Set Splitting is a transformation used to split the index range of a loop in order that fewer elements of an array are accessed. If the index range is larger than the page size, then index set splitting can be used to improve I/O performance. Index set splitting can be a useful technique to improve I/O behavior.

3.14 Loop Peeling

Loop peeling is a technique used to remove anomalies in the control flow of a loop. Loop peeling is a technique similar to index set splitting, but unlike index set splitting. loop peeling is not a useful technique if one wants to improve I/O performance. Index set splitting can be applied to the inner loops, which may reduce memory requirements; loop peeling, on the other hand, involves both the outer and inner loop to break up a loop. Since both loops are used, less of each variable is used, which leads to less efficient use of resident variables, which leads to more page transfers. Therefore loop peeling is not a useful technique to improve I/O behavior.

3.15 Loop Unrolling

Loop unrolling is a technique where copies of the loop body are made so that the loop overhead can be reduced. It is not useful to improve I/O performance because it does not reduce memory requirements nor does it change the access pattern to match the storage pattern of arrays.

4. THE ALGORITHM

We outline the I/O performance improvement algorithm (IOPIA). We have already determined that the code transformation techniques useful to improve I/O performance are loop distribution, loop interchange, replication (variable copying), node splitting, multiversion loops, scalar expansion, and index set splitting. The primary goal is to apply loop interchange because it is the transformation which will provide the most I/O performance improvement. Loop interchange cannot be applied when there exists an interchange preventing dependence (IPD) in the loop, or if there is an order of indices error (OIE), i.e., the indices for A(I,J)=A(J,I) do not match. No transformation will eliminate an OIE while loop distribution is the only transformation that can eliminate an IPD, by distributing the statements so that the IPD no longer exists in the loop. Cyclic dependences will cause loop distribution to fail; to break a cyclic dependence we apply the transformations replication, multiversion loops and scalar expansion. If the IPD cannot be broken, we use replication, multiversion loops, scalar expansion and node splitting to distribute as much of the loop away from the IPD so that some of the loop might be interchanged. IOPIA can be broken down into two main cases: processing a single loop and processing a nested loop. In Section 4.1 we explain how the IOPIA improves the I/O behavior of a single loop. Section 4.2 applies IOPIA to a nested loop. Section 4.3 discusses the criteria ensuring that the transformations preserve the semantics of the given code. Section 4.4 gives the pseudocode for the IOPIA, Section 4.5 sketches its time complexity.

4.1 Single Loop

Loop interchange cannot be applied in a single loop. Single loops cause few page transfers compared to nested loops, unless the amount of available memory is very limited. Thus, little improvement can be expected from transforming a single loop. For single loops, IOPIA checks if the memory required by the loop exceeds the available memory. If so, IOPIA first attempts to apply loop distribution to reduce the memory requirements; otherwise, it tries to apply multiversion loop, variable copying, scalar expansion and node splitting to break the cyclic dependences that caused loop distribution to fail. Once the memory requirements of the loop is reduced so that the available exceeds the required memory, IOPIA terminates.

4.2 Nested Loops

Nested loops are where the most I/O improvement are made. Before any transformations are applied, the IOPIA checks whether the memory required by the loop exceeds the available memory. If so, IOPIA processes the loop. The goal of IOPIA is to apply loop interchange to a nested loop. The only time when interchange cannot be applied is when there exists an IPD in the loop body or when an OIE is detected. If an OIE exists, the loops cannot be interchanged. The IPD can occur in two different situations; the IPD is either a dependence from a statement onto itself (Type A; IPD-TA), or a dependence between two different statements (Type B; IPD-TB). For the case IPD-TA, the only transformation technique which might be able

to break the dependence is multiversion loops. Here is the method IOPIA uses to process a nested loop containing an IPD-TA:

Step 1: If applicable, apply multiversion loops to break the IPD
Step 2: If possible, distribute the loop to separate statements from the IPD
Step 3: If any statements were not separated from the IPD, then apply the appropriate transformation(s) (multiversion loops, variable copying and scalar expansion) in order to distribute statements away from the IPD
Step 4: If any statements still remain bound to the IPD, apply node splitting (if applicable) in order to separate variables away from the IPD

After Step 4, the algorithm will examine the output. If the performance of the transformed code is worse than the original, the algorithm will revert back to the original code.

For the case IPD-TB, the dependence may be broken by distributing the two statements into different loops. Two statements that have an IPD between them can always be distributed, unless they are part of a dependence cycle. Here is the method that IOPIA uses to process a nested loop containing an IPD-TB:

Step 1: If possible, apply loop distribution to break the IPD
Step 2: If distribution failed, apply the appropriate transformation(s) (multiversion loops, variable copying, and scalar expansion) in order to distribute the statements and break the IPD
Step 3: If the IPD cannot be broken, distribute the loop in order to separate statements away from the IPD
Step 4: If the IPD cannot be broken and some statements could not be separated from the IPD, apply the appropriate transformation(s) (multiversion loops, variable copying and scalar expansion) in order to distribute statements away from the IPD
Step 5: If the IPD cannot be broken and some statements still remain bound to the IPD, apply node splitting (if applicable) in order to separate variables from the IPD

4.3 Transformation Techniques

For each of the techniques involved (multiversion loop, variable copying, scalar expansion, index set splitting), we can formulate criteria that assure that the code resulting from applying the corresponding code transformation is semantically valid (i. e., the new code computes the same results as the original code). Because of the page limit, we refer to [T96] for more information about these criteria.

4.4 IOPIA Pseudocode

The algorithm applied to any arbitrary set of statements $S_1,...,S_n$ in a loop L will either provide some improvement to the I/O behavior of the loop, or in the worst-case, leave the loop unchanged. We collect information from the source code and organize it into three tables ([H95, T96]). The first structure is the Instruction Table. It contains essentially syntactic information about the program statements. The second is the Array Table which holds syntactic data specifically about arrays. The final structure is the Index Table which holds the data about the array indices involved. Because of the page limit, we refer the reader for details on the deta structures used in the algorithm to [T96].

Here is the pseudocode for the algorithm IOPIA; a comprehensive implementation of this algorithm is currently under way:

General Approach

1. Scan the program to build the tables.
2. Build the dependence graph and enter the data into the Instruction Table
3. If the memory used in the loop is less than the amount of available memory, then goto Step 9
4. If the index table contains more than one loop id, then goto Step 6
5. Distribute the loop
   5.1. if distribution fails, or if the distributed code still requires more memory than is available, apply multiversion loops, variable copying, scalar expansion and node splitting in order to distribute further
   5.2. goto Step 9
6. if the Instruction Table does not contain an IPD or an OIE, then interchange the loop and goto Step 9
7. if the Instruction Table contains an OIE, goto Step 8.2
8. if the Instruction Table contains an IPD
   8.1. If the IPD is Type B, then distribute
      8.1.1. if distribution fails to break the IPD, apply multiversion loops, variable copying and scalar expansion in order to distribute and break the IPD
      8.1.2. if the IPD is broken, apply interchange and goto Step 9
      8.1.3. if the IPD still exists, then goto Step 8.2
   8.2 If the IPD is Type A, then apply multiversion loops to try to break the IPD
      8.2.1. if multiversion loops failed, then distribute

8.2.2. if some statements still exist in the same loop as the IPD, apply multiversion loops, variable copying and scalar expansion in order to distribute the statements away from the IPD
8.2.3. if some statements still exist in the same loop as the IPD, then apply node splitting
8.2.4. if we have made things worse, then undo all changes and goto Step 9
9. return the result

Multiversion Loops Algorithm:
1. if the applicable criterion holds
   1.1. find the values for the runtime value that break the dependence
   1.2. duplicate the entire loop
   1.3. prefix each loop with an if statement for the respective values of the runtime variable
   1.4. distribute the loop which contains the values that do not contain the dependence

Variable Copying Algorithm:
1. if the applicable criterion holds
   1.1. create a new variable and new assignment statement
   1.2. insert the new assignment statement before both statements of the dependence cycle
   1.3. modify the statement to use the new variable
   1.4. distribute the loop

Scalar Expansion Algorithm:
1. if the applicable criterion holds
   1.1. change all occurrences of the scalar variable into an array variable
   1.2. distribute the loop

Node Splitting Algorithm:
1. if the applicable criterion holds
   1.1. create a new variable and assignment statement
   1.2. insert the statement textually before the two statements which are involved in a dependence cycle
   1.3. modify the statement to use the new variable
   1.4. distribute the loop

4.5 Brief Time Complexity Determination of IOPIA

The IOPIA algorithm is used at compile time, not at run time; thus the time complexity of the algorithm depends on the number of statements in the loop, but not the amount of data. The input to the algorithm is a loop L of statements $S_1,...,S_n$ (of fixed complexity) where n is the number of statements in the loop. Step 1 requires O(n) time. Step 2 requires $O(n^2)$ time. Step 3 is O(1). Step 4 requires O(n) time, while Step 5 takes O(1) time, as does Step 6. Step 7 requires $O(n^2)$. Thus, the overall time complexity of the IOPIA algorithm is $O(n^2)$ where n is the size of the code. Most importantly, IOPIA's time complexity does not depend on the size of the data.

5. DISCUSSION OF RESULTS

We present examples demonstrating the effectiveness of the code restructuring algorithm. We assume that the storage scheme is column major, the working set size WS and page size PS are given as input, and the replacement strategy used for virtual memory management is (pure) LRU (Least Recently Used).

Example 5.1 with WS=2, PS=1024

```
Do I = 1, 2048
   S1: A(I) = B(I) + C(I)
   S2: E(I) = F(I) + G(I)
   S3: B(I) = C(I) * 2
   S4: F(I) = G(I) * 2
   S5: C(I) = (A(I) + B(I)) / C(I)
   S6: G(I) = (E(I) + F(I)) / G(I)
End Do
```

IOPIA distributes the loop into the following:

```
Do I = 1, 2048  S1  End Do
Do I = 1, 2048  S3  End Do
Do I = 1, 2048  S5  End Do
Do I = 1, 2048  S2  End Do
Do I = 1, 2048  S4  End Do
Do I = 1, 2048  S6  End Do
```

Four of the blocks still require more memory than is available, but IOPIA cannot apply any other transformations to them. With the working set size equal to 2, the original code requires 32,768 page transfers, while the transformed code requires 24,582, an improvement of 24.98%.

Example 5.2

```
Do I = 1, 2048
   S1:  A(I) = B(I) + C(I) * D(I)
   S2:  D(I) = B(I) - C(I)
   S3:  B(I+1) = A(I) * C(I)
   S4:  E(I) = F(I) + G(I) * H(I)
   S5:  G(I+1) = E(I) + H(I)
   S6:  H(I) = G(I) + F(I)
End Do
```

Case 1 WS=6, PS=2048: IOPIA distributes the code resulting in:

```
Do I = 1, 2048  S1; S3  End Do
Do I = 1, 2048  S2  End Do
Do I = 1, 2048  S4; S5; S6  End Do
```

The original code requires 10,243 page transfers, while the transformed code requires 8, an improvement of 99.92%.

Case 2 WS=3, PS=2048: First IOPIA does loop distribution, but the memory required by the first and the last loop still exceed the available memory, so IOPIA does node splitting on the first loop:

```
Do I = 1, 2048
      T1(I) = C(I) * D(I)
End Do
Do I = 1, 2048
   S1':  A(I) = B(I) + T1(I)
   S3:   B(I+1) = A(I) * C(I)
End Do
```

The original code requires 28,672 page transfers, the transformed code 12,294, for an improvement of 57.12%. Thus, IOPIA continues to process loops until their memory requirements are reduced to below the available resources, or until no more transformations can be performed.

Example 5.3 with WS=3, PS=2048

```
Do I = 1, 2048
   S1:  A = B(I) + C(I)
   S2:  B(I) = A * D(I)
   S3:  C(I) = B(I) + B(I+1)
   S4:  E(I) = C(I) * D(I)
   S5:  D(I) = A * A
End Do
```

After the initial distribution step of IOPIA, no change is made to the loop because nothing can be distributed. Thus, IOPIA applies two different transformations, variable copying and scalar expansion. Variable copying creates a new statement and modifies the statement S_3, resulting in (after distribution):

```
Do I = 1, 2048
   S0:  T1(I) = B(I+1)
End Do
Do I = 1, 2048
   S1:  A = B(I) + C(I)
   S2:  B(I) = A * D(I)
   S3':  C(I) = B(I) + T1(I)
   S4:  E(I) = C(I) * D(I)
   S5:  D(I) = A * A
End Do
```

Scalar expansion changes all of the occurrences of the scalar variable A into an array variable, resulting in (after distribution):

```
Do I = 1, 2048  S0  End Do
Do I = 1, 2048  S1  End Do
Do I = 1, 2048  S2  End Do
Do I = 1, 2048  S3  End Do
Do I = 1, 2048  S4  End Do
Do I = 1, 2048  S5  End Do
```

The original code requires 6,145 page transfers, the transformed code 11, an improvement of 99.82%.

Example 5.4

```
Do I = 1, 2048
   Do J = 1, 2048
     S1:  A(I,J) = B(I,J) + C(I,J) * D(I,J)
     S2:  B(I,J) = C(I,J) + D(I,J)
     S3:  C(I,J) = C(I,J) + C(I-1,J+1)
     S4:  E(I,J) = F(I,J) - G(I,J)
     S5:  F(I,J) = E(I,J) * E(I,J)
     S6:  H(I,J) = F(I,J) + G(I,J) * E(I,J)
   End Do
End Do
WS = 1000, PS = 1024
```

IOPIA detects an IPD-TA in the loop and tries to apply multiversion loops which fails to break the IPD. Then IOPIA tries to distribute the loop in order to distribute as many statements away from the IPD. All the statements can be distributed, so IOPIA distributes them and interchanges the loops, resulting in:

```
Do J = 1, 2048  Do I = 1, 2048  S1  End Do  End Do
Do J = 1, 2048  Do I = 1, 2048  S2  End Do  End Do
Do I = 1, 2048  Do J = 1, 2048  S3  End Do  End Do
Do J = 1, 2048  Do I = 1, 2048  S4  End Do  End Do
Do J = 1, 2048  Do I = 1, 2048  S5  End Do  End Do
Do J = 1, 2048  Do I = 1, 2048  S6  End Do  End Do
```

This example showns how to us loop distribution in conjunction with loop interchange. The original loop cannot be interchanged because of the interchange preventing dependence in statement S_3. But after applying loop distribution, the statements are broken up into individual loops which allow the other five statements to be interchanged. The original code requires 33,558,528 page transfers, the transformed code 4,263,936, an improvement of 87.29%.

6. CONCLUSION

We presented an algorithm that improves the I/O behavior of a given program by applying code transformations to it. These are loop distribution, loop interchange, multiversion loops, variable copying, scalar expansion and node splitting. We demonstrated the power of our algorithm by several examples and showed that it can greatly improve a program's I/O performance. Future enhancements include improving the analysis of the Instruction Table so that one can tell if a particular transformation will improve performance or not, as well as improving the dependence analysis so that we can determine whether interchange or index set splitting should be applied, if an order of indices error occurs.

7. BIBLIOGRAPHY

[C84] R. W. Carr, *Virtual Memory Management*, UMI Research Press, Ann Arbor, Michigan, 1984.

[H95] Y. H. Hseu, *Automatic I/O Behavior Improvement Through Loop Interchanges*, M. S. Thesis, Department of Computer Science, University of Houston, 1995.

[L85] B. R. Leasure, *The Parafrase Project's Fortran Analyzer. Major Module Documentation.* Technical Report CSRD-504, Department of Computer Science, University of Illinois at Urbana-Champaign.

[L95] E. L. Leiss, *Parallel and Vector Computing: A Practical Introduction*, McGraw-Hill, Inc., 1995.

[PW86] D. A. Padua and M. J. Wolfe, *Advanced Compiler Optimizations for Supercomputers*, CACM, December, 1184-1201, 1986.

[T72] R. E. Tarjan, *Depth First Search and Linear Graph Algorithms*, Computing, 1, 1972.

[T96] W. C Tu, *I/O Performance Improvement Through the Use of Code Transformations*, M. S. Thesis, Department of Computer Science, University of Houston, 1996.

[W82] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph.D. Dissertation, Technical Report 82-1009, Department of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[ZC91] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley, New York, NY, 1991.