A SOFTWARE TOOL FOR DATA TRANSFER ANALYSIS
AT COMPILATION TIME IN VIRTUAL MEMORY SYSTEMS

E. L. Leiss, L. Raffoni, and W. W. Ou
Department of Computer Science
University of Houston
coscel@cs.uh.edu

Abstract
Since storage requirements continue to exceed the capacity of the computer memory, data transfer analysis is important for any program designed to deal with a large amount of data. Therefore, programmers must know the I/O behavior of their software and possibly use this knowledge to restructure it in order to obtain better performance. To this end, we designed an algorithm which, at compile time, computes an approximation on the number of data transfers, between main and secondary memory, necessary during the execution of a given FORTRAN program. The underlying environment is a virtual memory system applying the Least Recently Used replacement policy. To perform this analysis, we must make several assumptions and simplifications, so the result we obtain is an estimate. In all cases, the obtained approximation is an upper bound on the actual number of data transfers. Regardless of the error in the result (often unavoidable at compile time), our tool is useful for comparing different versions of a given algorithm.

1. Introduction
In recent years, while high performance programs have become more and more memory demanding, the increase in physical memory has often not followed suit and has frequently proven insufficient. A solution to this problem requires one to extend somehow the capacity of the system memory without enlarging it physically, which is exactly what virtual memory systems allow one to do.

The main problem of these systems stems from the fact that blocks of information need to be transferred between secondary and main memory during execution of a program. This operation requires particular attention, since it is expensive and its uncontrolled repetition reduces substantially the overall performance. Several replacement strategies have been studied [C84] to obtain the best possible choice of the page to move out when some space in main memory is needed and no free blocks are available. Nowadays good approximations on future system behavior are available; however, sometimes the programs themselves need to be restructured to obtain a better use of the data and the possibility to further reduce the number of system I/O operations. A global description of the data paths that were taken during program execution (I/O profile) can therefore be very important in deciding whether the program is efficient from the I/O point of view or not [L90].
We created a tool to do such an analysis automatically, based on the work initiated in [LO91]. Working with simplified FORTRAN programs, our analyzer reports a compile time approximation of the number of data transfers the program will need for execution, based on the parameters of the system currently available. As we will see later, some approximations and restrictions apply, but the final result can be considered as reflecting the real behavior of the system when executing the program and can therefore be used as a parameter for future comparisons, in case some restructuring of the program is carried out or a different system is used.

If the program does require too much I/O work, restructuring becomes imperative as shown by several studies addressing the issue of program restructuring to minimize the data transfers between main and secondary memory. Most of the previous work emphasizes module rearrangement for optimizing the locality of module references. In [HG71], for example, Hatfield and Gerald presented a method directed to make contiguous in memory those modules that most frequently reference each other. This method is independent of the underlying memory management policy. A different approach, called strategy dependent policy, was taken in [F74a], [F74b], [F75], [F76a], [F76b] and [P84]; it models the page replacement policy of the operating system on the base of the program to be executed. Ferrari et al. carried out many experiments on performance improvement through program restructuring and obtained encouraging results both when each process is treated as a separate entity and in real time sharing environment.

The paper is organized as follows. In Section 2 we present an overview of the main ideas and features underlying our analyzer; Section 3 explains how to use our tool and Section 4 how the analysis is implemented. In Section 5, the final algorithm described. Section 6 reports our conclusions.

2. Overview of the I/O Analyzer

Our algorithm is designed to calculate, at compile time, a reasonably accurate estimate of the total number of data transfers required between main and secondary memory when executing a given FORTRAN program. To make possible an automatic analysis of the data transfers, we decided to work on simplified versions of the original FORTRAN programs, based on the fact that they handle a large amount of data, typically stored in matrices and referenced inside DO loops. Therefore we assume the following simplifications:

- only the DO loops of the target program are analyzed; all the other parts are simply skipped (this is because the loops are considered the main source of data transfers);

- only array references are considered when looking at the items to be transferred from secondary to main memory and vice versa.

The final result gives the number of blocks of data transferred between main and secondary memory when the Least Recently Used replacement strategy is applied.

The analyzer we implemented is based on the values of some parameters related to the memory allocation: without such information the analysis cannot be performed as it must necessarily give different results in different environments. At the same time, to have the exact values of these parameters is very difficult at a user or programmer level, or even impossible at compile time; therefore an acceptance of some hypotheses is necessary.
2.1 System Assumptions

The system represents the environment in which the execution of the target program takes place; we assume it has a single process accessing its own memory, or part of it, without sharing those addresses with other processes.

Every compiler transforms multidimensional arrays to store them in a one-dimensional vector following either row or column major order. We assume that this one-dimensional vector, in which any array is transformed for storage purposes, is stored sequentially in successive pages starting from the beginning of them. Furthermore, we assume that no page is shared between different pieces of data.

The page replacement policy assumed is the Least Recently Used strategy (LRU), after which almost all replacement strategies currently in use are patterned.

2.2 Program Assumptions

The program assumptions can be divided into two different kinds: those stated at the beginning, simplifications of the real situation represented by the program we want to analyze, and those we impose during the analysis itself, when there are special features which cannot be solved by the normal analysis. What characterizes all of the latter ones is that we always assume a worst case scenario. This means that whenever we face special situations, we assume the nearest one which we can handle and whose number of transfers represents an upper bound on those of the case at hand.

The first group of assumptions is mainly reflected in the way we carry out the analysis; its main feature is the restriction to the DO loops among all the possible statements present in a FORTRAN program. Further assumptions apply to the treatment of branches, IF and WHILE statements which can be included in a DO loop body. For the IF statement we again apply a worst case assumption and suppose that the body of the IF is always entered and the branch never taken. In case we have an IF THEN ELSE construct, since it is impossible to predict which part will be entered during execution (and most probably this choice won't always be the same for each iteration), we count as referenced all the array instances both occurring in the THEN and in the ELSE branch. Even if this is an impossible case in the real world, we follow it to satisfy the worst case assumption strategy.

For WHILE statements it is generally not possible to calculate at compile time how many times the body is entered or even an upper bound on this number. Therefore we assume that: the body of any WHILE loop is always executed exactly once. In this way the arrays referenced there are brought into memory (worst case) but no assumption about the number of times they are accessed is made. It should be noted that this is a violation of our guiding principle, namely to assume always a worst case scenario. However, this is unavoidable since it is impossible in general to predict the number of iterations of a general WHILE loop at compile time. It was felt that this assumption was preferable to an outright prohibition of WHILE statements.

2.3 Data Transfer Analysis Method

The method we used to determine the total number of data transfers refers to the one presented in [LO91] whose steps are here summarized. The main idea consists of applying a sequence of reductions. Here are the reduction steps:
1. Reduce the data transfer in a program to the sum of data transfers in each multilevel loop;
2. Reduce the data transfers in a multilevel loop to the sum of data transfers in the innermost loop (assuming no branch is taken inside it);
3. Reduce the data transfers in an innermost loop to the sum of data transfers for each non-equivalent page reference sequence;
4. Reduce the data transfers in a page reference sequence to the sum of its data transfers to set up the sequence initially plus the data transfers to execute the iterations until a page move occurs.

**Definition 2.3.1.** An array reference sequence (ARS) $S = (R_1, R_2, ..., R_n)$, referring to a particular DO loop, represents the ordered list of the arrays accessed in the DO loop body; the order in which they appear in the list must reflect the order of reference.

For example, if the body consists of:
$A(I) = B(I) + C(I)$
$C(I) = 0$
then $S = (B, C, A, C)$, since the operands are always accessed before the variable containing the result of a mathematical operation.

**Definition 2.3.2.** A page reference sequence (PRS) $P = (R_1[P_1], R_2[P_2], ..., R_n[P_n])$ is an extension of the ARS associated with a DO loop body, where $R_i[P_i]$ denotes the $P_i$th page containing the array $R_i$.

As an example, assume that array $A(10,10)$ is stored in row major in pages of size 10 and is referenced in the following loop:

```
DO 20 I = 1, 5
  DO 10 J = 1, 5
    A(I,J) = A(I+1, J)
  10 CONTINUE
20 CONTINUE
```

The page reference sequence associated with the innermost loop during the first iteration of the outermost one is $(A[1], A[2])$; in general, when $I = i$, it has the form $(A[i], A[i+1])$. While the ARS of a DO loop remains constant, its PRS is continuously updated on the basis of the values of the array indices. The importance of the PRS management is obvious, since we need to look at it to know which pages have to reside in main memory in a certain iteration. Actually, the algorithm statically determines what will be the evolution of the initial page reference sequence and consequently determines the data transfers to update it successively.

**Definition 2.3.3.** Given a PRS $P = (R_1[P_1], R_2[P_2], ..., R_n[P_n])$, $R_i[P_i] = R_j[P_j]$ iff $R_i = R_j$ and $P_i = P_j$.

**Definition 2.3.4.** Two page reference sequences $P = (R_1[P_1], R_2[P_2], ..., R_n[P_n])$ and $S = (S_1[Q_1], S_2[Q_2], ..., S_n[Q_n])$ are equivalent if they satisfy:
1. For all $i = 1, 2, ..., n$, $R_i = S_i$;
2. For any $i, j = 1, 2, ..., n$, if $R_i[P_i] = R_j[P_j]$ then $S_i[Q_i] = S_j[Q_j]$.

The concept of "equivalence" of page sequences is important because of the fact that different amounts of data transfers may correspond to iterations with non-equivalent PRSs, while to perform iterations showing equivalent PRSs the transfers needed are the same. This, in particular, is true because the page replacement policy is LRU: in this case the
transfers occurring during the execution of a single DO loop cycle depend on the "repetitiveness" of the pages in the PRS itself. If two PRSs show in the same places a particular page (maybe different between the two PRSs), they will need the same transfers from secondary memory. This "having in the same positions the same page" is exactly what is formally translated in point 2 above.

Our algorithm identifies sequences of iterations showing equivalent PRS; for each of these phases it calculates the corresponding data transfers, which are finally summed up.

The determination of the data transfers corresponding to all the iterations presenting equivalent PRS is practically implemented by studying the evolution of the PRS itself, with respect to the modifications of the pages it contains. More precisely, this value can be decomposed into three terms to be summed up, corresponding to phases in the life of a (group of equivalent) PRS:

1- transfers due to the establishment of the first PRS and the corresponding initial set up of memory (b);

2- transfers due to changes of some pages referenced in the list, which don't cause the new one to be non-equivalent to the previous version (d);

3- transfers occurring during execution of a DO loop cycle (c).

The total amount of data transfers required for PRSs that remain equivalent (if not the same) for K iterations of the innermost DO loop to which they refer, finally is

\[ t = b - c + d + c \times K. \]

3. How to Use the Software Tool

3.1 The Program to Be Analyzed

The FORTRAN program the user wants to analyze represents the first parameter which has to be provided; in fact this will not be the program as it is designed for its "normal" execution, but a simplified version of it, which contains only the information significant for the analysis to be carried out. It is important that the program be presented to the analyzer after a FORTRAN compiler has checked it syntactically. Programs incorrect from a syntactical point of view sometimes cannot be recognized and in these cases an error message may be displayed.

More specifically, the simplified version to give as a parameter has to be structured in two parts: declarations and loops listing.

**DECLARATIONS**

The *declarations* part contains the usual FORTRAN declarations for the arrays which will be referenced later in the DO loop bodies. It is intended that an array not declared in this part won't be recognized and analyzed later.

The only restriction applying to the declaration of arrays is that their indices start from 1. This assumption is not very restrictive since any array can be transformed to an equivalent one with this characteristic.

391
LOOP LISTING
The loop listing part includes all and only the DO loops of the original program. There can be an unlimited number of DO loops one after the other; the analyzer will scan them in order and test for each of them whether they satisfy the following:

- The depth of any nest of DO loops is at most seven;
- Each DO statement has a positive step increment;
- At most 64 array references are present in the body of a loop;
- The expressions representing array indices are linear functions involving only the following operations: addition, subtraction, multiplication or division. These expressions may involve integers together with variables.

3.2 The System

In addition to the program the user must supply several system parameters. They are:

i - page size, in terms of the elementary data (words) it contains;
ii - working storage set size, in terms of the pages it contains;
iii - the way (row or column major) multi-dimensional arrays are mapped into memory.

Modifications of these values may alter radically the result, and with their control the user can compare how the FORTRAN algorithm he designed behaves on different systems.

3.3 How to Read the Results

As already pointed out, the result of our analysis represents an approximation of the total number of data transfers which occur when executing our target program, presented as explained in the previous section. This approximation is an upper bound on the real result and includes both the transfers of pages from main to secondary memory and vice versa.

The final result presented is the "total number of data transfers for the innermost loop alone". This means that the number of transfers is the one needed to execute ONLY the innermost loop. The requirement that the expressions of the array indices be linear functions of the loop variables ensures that the transfers are always the same for any value of the outer loop variables. Therefore, if several loops are nested, to obtain a rough estimate of the transfers needed for the entire nested loop, the user can multiply the result given by the program for the total number of times the innermost loop is executed.

A warning applies to the use of variables: since the analyzer cannot give a numeric value to any of the variables involved (even when this value may be clear to the programmer from the context) a heavy use of them may create undecidable situations. The analysis will be carried out nevertheless and a warning will displayed for the user, but the results should be considered with caution. Here is a list of the values that should be given as numeric values:

- start, end and step value in DO loop expressions;
- page size;
- working storage set size;
- array dimension specifications;
- operators involved in the array reference index expressions.
4. Implementation

Our algorithm can be separated into successive phases: initially the target program is scanned and the information needed to perform our calculations is extracted. These data are then examined and the result, reflecting the future behavior of the system, is obtained in three steps (b, d, c, see Section 2.3).

4.1 Collection of Data

The entire analysis is based on the progress of the PRS, whose content is continuously updated and checked. To track the evolution of the PRS we need some basic notions about the arrays referenced; we start with starting index and index increment. Both parameters refer to positions in the one-dimensional vector in which any multidimensional array is mapped in memory. The starting index (SI) represents the index of the first array element referenced in the loop body, while the index increment (II) counts the number of elements between two references of the same array in successive iterations of a certain innermost loop.

We assume that the following information was extracted from the program:

- \((D^1, D^2, \ldots, D^k)\) is the dimension specification of the array;
- \(k\) is the total number of dimensions of the array;
- \((j_0, j_1, \ldots, j_n)\) are the positions in array reference indices containing the innermost loop, such that \(1 \leq j_0 < j_1 < \ldots < j_n \leq k\). \(NN_i = ST * c_i\) for all \(i\) in \(\{j_0, j_1, \ldots, j_n\}\);
- \(c_i\)'s are the coefficients of the innermost loop variable for all \(i = j_0, j_1, \ldots, j_n\);
- \(I^j\) represents the initial value declared in dimension \(j\) of the array (which we assume to be always 1) and \(I^j\) is the value of the same index at the start of the loop;
- \(ST\) represents the step increment of the innermost DO loop variable.

Here are the formulas:

(i) Row Major:

\[
II = NN_{j_0} * D_{l_0} + 1 * \ldots * D_k + \\
+ NN_{j_1} * D_{l_1} + 1 * \ldots * D_k + \\
+ \ldots + \\
+ NN_{j_n} * D_{l_n} + 1 * \ldots * D_k;
\]

(ii) Column Major:

\[
II = NN_{j_n} * D_{l_n} - 1 * \ldots * D_1 + \\
+ NN_{j_{n-1}} * D_{l_{n-1}} - 1 * \ldots * D_1 + \\
+ \ldots + \\
+ NN_{j_0} * D_{l_0} - 1 * \ldots * D_1.
\]

Example 4.1.1. If \(A(100, 100, 100)\) is referenced in this DO loop:

\[
DO 20 I = 1, 100 \\
DO 10 J = 1, 50 \\
A(I, J, J*2) = 0 \\
10 CONTINUE \\
20 CONTINUE
\]

the succession of \(A\) elements referenced along the innermost loop is \(A(i, 1, 2), A(i, 2, 4), \ldots, A(i, j, j*2), \ldots, A(i, 50, 100)\), where \(i\) is any value between 1 and 100. We can see that the positions of the array indices containing the innermost loop variable are 2 and 3; therefore \(j_0 = 2, j_1 = 3\), \(NN_2 = 1\) and \(NN_3 = 2\). In the case of row major storage pattern,
between two generic elements \( A(i, j, j^2) \) and \( A(i, j+1, (j+1)^2) \) referenced in successive iterations there are the following elements:

\[
A(i, j, j^2+1), \ldots, A(i, j, j^2+k), \ldots, A(i, j, 100),
\]

\[
A(i, j+1, 1), A(i, j+1, 2), \ldots, A(i, j+1, (j+1)^2)
\]

which correspond to \( 102 (= 100 - j^2 + (j+1)^2) \) total elements; applying formula (i) we obtain:

\[
II = NN_2 * D^3 + NN_3 = 1 * 100 + 2 = 102.
\]

SI and II are used to obtain F and M, the iteration in which the array reference changes page for the first time and the frequency at which its changes of page take place, respectively:

* page move frequency: \( M = P / II \);
* iteration of first page move: \( F = (P - "" SI) / II \);

where \( P \) represents the size of a page. The operation \( P - "" IDX \) gives as a result the amount of data between the array element in position IDX of the one-dimensional vector (in which the array is mapped for storage purposes) and the end of the page in which it is stored:

\[
P - "" IDX = P - (IDX \ mod \ P).
\]

Using F and M we can establish at compile time in which iterations the array element they refer to is not residing anymore in the same page of the previous cycle. This means that for those iterations we need to transfer a new page from secondary memory, and possibly transfer the replaced page from main to secondary memory.

### 4.2 Algorithmic Details

The determination of the total number of data transfers follows some steps based on the evolution of the PRS corresponding to the loop we are analyzing. In this paragraph we present the global algorithm applied by our analyzer to a (nested) DO loop. We assume that all the PRSs generated during the loop life time are equivalent, therefore it suffices to determine the transfers needed for the first generated PRS and extend the resulting value to the entire DO loop.

#### 4.2.1 Transfers During the Execution of a Single Iteration

The transfers occurring between main and secondary memory in a loop cycle (c) can be easily calculated by adapting an algorithm designed by Peterson (P86) to the LRU policy. Here are the main steps, where the memory is considered as a queue ordered from the last (top) to the first (bottom) page transferred to it. Parameters of this function are:

- \( S \) = current PRS;
- \( WS \) = working storage set size;
- \( IS \) = PRS referenced in the iteration preceding the present one.

**Algorithm A(S,WS,IS)**

1. Set the queue \( Q \) to empty;
2. If \( IS \) is not empty, fill in the queue \( Q \) with the last \( WS \) pages (coming from the previous iteration) of \( IS \) and set \( c=0 \);
3. Scan \( S \) from the first page \( P_1 \) till \( P_N \); for each \( P_i \):
   a. if it is in \( Q \) move it to the first position, shifting to the bottom all the pages before it;
   b. otherwise transfer \( P_i \) to the top, shift by one position all the following elements and remove the bottom page from memory; add 1 to \( c \);
4. Return \( c \).
4.2.2 Page Reference Sequence Evolution

As pointed out, when the PRSs are all equivalent the number of transfers is the same for each iteration, even though some pages may change in the mean time. Creation of and successive modifications to a PRS determine the other terms to be added to get the final number of data transfers.

Algorithm A is applied both to determine b and c. When we use Algorithm A to determine b (the transfers to set up the initial PRS), the list of pages previously referenced has to be empty, since the PRS is now being installed for the first time.

During iterations in which the PRS is updated with respect to the previous one, additional transfers may be needed in order to bring the new pages into the memory. However, depending on the positions in the PRS of the updated pages, there is the possibility of having the same number of data transfers we have in iterations referring to the same PRS of the previous one. The number of transfers due to page moves is defined as:

\[ d = \sum_i \left( N - F_i \right) / M_i, \]

where the sum is taken over all i such that R \(_i\)'s page, if not changed, is in memory from the previous iteration at the time in which it must be referenced. M \(_i\) represents the page move frequency and F \(_i\) the iteration of first page move of array reference R \(_i\).

To recognize whether an array's change of pages adds new transfers to the ones already calculated in c, we use the following Algorithm B. Its output is the list D of the arrays to consider in the calculation of d; the parameters are as follows: S is the initial PRS, A is the set of arrays in the ARS and WS denotes the working storage set size. S[r] is the rth element in S; A[r] indicates the array in A corresponding to S[r].

Algorithm B(S, WS)
1. If the number of unique pages in S (U) is less than or equal to WS, D = A;
2. Otherwise: Mem contains the last unique WS elements in S; r = 1; m = 1; D = {};
   repeat
      a- if S[r] is equal to one of Mem[m], ..., Mem[WS] then insert A[r] in D; else
      b- if S[r] is also not in Mem[0], ..., Mem[m-1] then Mem[m] = S[r] and m = m+1;
      c- r = r + 1;
      until (r = <total elements in S>+1 OR m = WS + 1);
3. Return D.

5. The Final Algorithm

There are several additional considerations which influenced the design of our algorithm, in particular the treatment of nested loops and the overlap of array references. We sketch some of these below; for more details, we refer to [R93].

**Definition 5.1.** Two array references are said to be overlapped iff, during execution of an infinite number of iterations of the loop, they are contained in the same page an infinite number of times (iterations).

From now on we will always consider two overlapping references, but all we will discuss can be extended to overlap involving more than two occurrences. Assume as usual:
- A\(_1\) and A\(_2\) are occurrences of the same array A;
- SI\(_1\) and SI\(_2\) are the starting indices of A\(_1\) and A\(_2\);
- II\(_1\) and II\(_2\) are the index increments of A\(_1\) and A\(_2\).
Proposition 5.2. Two array references A₁ and A₂ overlap if all these conditions hold:
(1) II₁ = II₂ = II;
(2) |SI₁ - SI₂| < P and SI₁, SI₂ refer to elements in the same page;
(3) P is divisible by II.

For each group of overlapping arrays we define its repeating cycle \((T = P / II, \text{where } II \text{ is the step increment size, equal for all references in the overlapping group); every } T\) iterations, the page situation relative to the arrays in the group repeats exactly, therefore it is enough to determine the data transfers in this period of time and to extend the result for the entire loop life time.

When more than one overlapping group is involved we have to consider what is going on step after step for all of them simultaneously. Since the repeating cycles are generally "intersecting" with each other, we have to consider the concurrent page modifications for all the groups. In this case the local repeating cycles \((T_i \text{ for each overlapping group})\) have to be replaced by a global repeating cycle \((T = \text{l.c.m.}\{T_i\})\).

Here then is the final algorithm, obtained after considering these issues:

**ALGORITHM**

1. Scan the program to obtain the following information:
   From the declarations part:
   • For each array \(R_i\) its dimension specifications;
   • total number \(m\) of arrays referenced;
   From the DO loops part:
   • ARS = \((R_1, R_2, ..., R_n)\) and for each array reference its indices expressions;
   • U = number of unique arrays in ARS;
   • total number of iterations of the innermost loop \(N\);
   • DO loop indices information (starting and ending values, steps);
   
   For each DO loop:
   2. Process these data; for each array reference define:
      • SI and II;
      • page move frequency \(M = P / II\);
      • iteration of first page move \(F = (P \text{ "-" } SI) / II\);
   3. Define \(PRS = (R_1[P_1], R_2[P_2], ..., R_m[P_m])\), where \(P_i = \lfloor SI_i / P \rfloor + 1\);
   4. Calculate \(b\), the total number of data transfers for initially setting up the PRS:
      • If \(U \leq WS\), \(b = U\);
      • otherwise call Algorithm A (see below): \(b = A(PRS, WS, \text{empty})\);
   5. For each array \(R_i\), if there is a previous array \(R_j\) in the ARS s.t. SIᵢ and SIᵢ are in the same page and \(II_j = II = II\), insert \(R_i\) and \(R_j\) (if it is not already there) in the same overlapping group.
   6. Determine \(d\), the total transfers due to page updates in the PRS:
      • For all \(R_i\) \((i = 1, ..., m)\) if \(F_i \geq N\) then \(dd_i = 0\) else \(dd_i = (N - F_i) / M_i\);
      • \(d = \sum_i dd_i\); the sum is extended for all \(i\) s.t. \(R_i\) is not included in any overlapping group and \(R_i\)'s page, if not changed, is in memory from the previous iteration;
If the number of overlapping groups is at least 1, execute the following code:

7. For each overlapping group \( G_1, ..., G_r \)
   * define \( T_i = P / I_i \);
   * define \( OL_i = \) list of the \( F_j \) for all \( R_j \) in the overlapping group;
8. Calculate \( T = \text{l.c.m.}\{T_i, i = 1, ..., r\} \);
9. Set \( it = 0, k = 0, PRS' = PRS \); in the following \( n \) is any integer;
   repeat
   * Check every \( F_j \) in \( OL_i \), for all \( i = 1, ..., r \), to find \( ac \) equal to
     a - \( \min\{n^*k+F_j : T \geq n^*k+F_j > it\} \), if it exists;
     b - \( T \), if \( \{n^*k+F_j : T \geq n^*k+F_j > it\} = \emptyset \);
     \( c_{ck} = ac - it \);
   * Define \( c_k' = A(PRSC', WS, PRS) \);
   * Calculate \( c_k \), the number of data transfers to execute an iteration:
     - if \( U \) is less than \( WS \) then \( c_k = 0 \);
     - otherwise \( c_k = A(PRSC, WS, PRS) \);
   * Define \( tck \) equal to
     a - \( c_k' + c_k * (c_{ck} - 1) \), if \( (c_{ck} - 1) \geq 0 \);
     b - 0, if \( (c_{ck} - 1) < 0 \);
   * \( PRS' = PRS \);
   * Update \( PRS \) changing the pages referenced by all \( R_j : n^*k+F_j = ac \);
     * it = \( ac \); \( k = k +1 \);
   until \( \{n^*(k-1)+F_j : T \geq n^*(k-1)+F_j > it\} = \emptyset \)
10. Return the total number of data transfers:
    \( t = b - c_0 + d + N/T \times \sum_{i=0,...,k-1} t_{ci} \);
Else (no overlap):
11. Calculate \( c \), the number of data transfers to execute one iteration:
    * if \( U \) is less than \( WS \) then \( c = 0 \);
    * otherwise \( c = A(PRSC, WS, PRS) \);
12. Return the total number of data transfers, \( t = b - c + d + N \times c \).
6. Conclusions

We presented an algorithm designed (see [LO91]) to compute at compile time an approximation of the number of pages transferred between main and secondary memory during the execution of a FORTRAN program. The fact that this analysis is performed at compile time implies that several parameters are not known and some simplifying assumptions have to be accepted from the outset. The result obtained represents the behavior of the program in the given system and offers a valid parameter to compare subsequent versions of the same algorithm.

Some improvements may be made on this algorithm to make it more precise. First of all, an error is introduced because of the fact that we separate the different situations involving the main memory and then study them separately. Sometimes this process causes us compute again some transfers already counted in other steps. To avoid this error an extension to our algorithm may try to recognize if some transfers have already been counted in some other term and avoid counting them again. However, this approach would introduce many other parameters.

Other criteria for overlap may also be studied. The one we used is somewhat restrictive; other approaches could be explored to overcome its limitations. For example, an extension of the overlap condition might consider also the number of iterations of the DO loop under study. This might avoid the case in which no overlap is assumed because we find out that it will not hold from a certain iteration on, but in fact it is valid during the entire DO loop lifetime. An even more sophisticated version could decide in which iterations the overlap holds and for which references, but this would require a different approach to the problem.

When designing our algorithm we took into account only the transfers between main and secondary memory. All architectures designed and produced in the last years actually introduce one more level in the memory system, that is they have a cache memory between the processor and the main memory. It would be interesting to explore the possibility to extend our algorithm in such a way that it considers the transfers occurring between both these levels. Having the possibility to define the parameters for the memory in the highest level of this hierarchy, the user can manipulate them appropriately and simulate the transfers between cache and main memory with this same algorithm.

BIBLIOGRAPHY


[F75] D. Ferrari, *Tailoring Programs to Models of Program Behavior*, IBM J, Res. Dev. 19, 1975, 244-251


