Automatic program transformations for virtual memory computers

by W. ABU-SUFAH, D. KUCK and D. LAWRIE

University of Illinois at Urbana-Champaign
Urbana, Illinois

INTRODUCTION

Improving the behavior of virtual memory systems is a popular subject, as evidenced by the vast number of papers in the literature. Typically, attempts to improve behavior fall into two areas—those which accept existing locality properties of programs and attempt to modify system parameters (e.g., memory allocated, window size for the working set policy, etc.), and those which attempt to reorganize programs in some way. The first approach treats programs behavioristically, i.e., without any attempt to change the original behavior of the program. This type of research generally attempts to deal with space allocation policies and replacement algorithms in order to improve the performance of the system, given the original behavior of the programs. The work of Denning,12,13 Belady,8 Chu and Opderbeck,9 Smith,21 Trivedi22 and many others has contributed greatly to the evolution of operating systems and hardware for virtual memory systems.

Yet the second approach promises even more significant improvement. Early work in this area11 indicated the importance of program reorganization, and more recent research (e.g., References 17 and 15) has borne out this promise. Our work belongs to this latter group, but with an important difference. We reorganize programs—automatically—by examining their structure at the source code level where more information about the program is available. The papers by Elshoff14 and Trivedi23 describe some similar techniques but left the restructuring to the programmer.

In the next section of this paper we present a brief description of our program transformations. By detailed analysis of the program, we reorganize the loop structure of programs in an attempt to ensure that once a page is used, as much computation as possible is done on that page before it is discarded and replaced by a new page. The result of our transformations is a program whose locality is better controlled. Our presentation will be through examples. For a formal description of the transformations, implementation problems and theorems related to the correctness of the transformations see References 3 and 1.

In the third section we present a summary of some preliminary experimental results which we obtained by applying our transformations to a collection of FORTRAN programs. As we will describe in that section, we have obtained good results so far in our work. We have seen more improvement in space-time product over standard paging from our source level transformations than we see in going from a nonpaging system to paging.

In the final section we will present some concluding remarks.

BRIEF DESCRIPTION OF THE TRANSFORMATION

Throughout this paper we will be concerned only with data paging. Moreover, we will ignore references to scalar variables. Similar assumptions were made by other researchers.6,5 The storage of each array will start on a page boundary. Moreover, we are primarily concerned with scientific programs; it is usually the programs with large arrays which cause serious problems for virtual memory computers.14

One of our principal transformations is distribution of DO-loop control. In order to see how this improves paging behavior, consider Program 1.

Program 1

```
DO 1 I=1, N
   A(I) = B(I) + C(I)
   X(I) = A(I) * X(I-1) + D(I)
   E(I) = 2 * X(I) + F(I)
1
```

Consider first a non-paged versus a paged machine. If allotted seven data pages, this loop will run completely through each of seven array partitions between page fault bursts. If each array occupied a total of \( p \) pages, then about \( 7p \) page faults would be generated in total, whereas a nonvirtual memory machine would have to allot a total of \( 7pz \) words of memory to its execution (\( z \) is the page size, in words). Thus, the memory-saving due to paging is a factor of \( p \). Generally, however, paging will increase the I/O activities over a non-paged system, because some pages may be refetched several times. Note that in programs containing

---

* This work was supported in part by the National Science Foundation under Grant Nos. MCS76-81686 and MCS77-27910, and the Department of Computer Science, University of Illinois at Urbana-Champaign.
eral loops, the same space can be used for each loop so the total space saved by standard paging is proportional to the product of $p$ and the number of loops in the program (assuming that distinct loops reference distinct pages).

**Program 2**

```
DO 1 I=1, N
  A(I) = B(I) + C(I)
DO 2 I=1, N
  X(I) = A(I) * X(I-1) + D(I)
DO 3 I=1, N
  E(I) = 2 * X(I) + F(I)
```

Suppose that Program 1 is rewritten as Program 2 by loop control distribution. This version can be run with an allotment of only three pages and still not generate page faults, except at the end of processing each set of pages. Note that in this case loop control can be distributed down to the level of individual statements.

**Program 3**

```
DO 1 I=1, N
  X(I) = Y(I-1) + A(I)
  Y(I) = X(I) * Y(I-1) + B(I)
```

In general, loop distribution is possible down to the level of cyclic data dependence graphs (called $\pi$-blocks) for individual loops. The nodes of a cyclic data dependence graph (each node representing a statement) are connected by directed arcs which form a cycle. An arc is drawn from one node to another if, at some instance, the statement represented by the first node must be executed before the statement of the second node. Thus, the loop of Program 3 cannot be further distributed since its two statements form a data dependence cycle and hence constitute a single $\pi$-block. Since such cycles seldom contain more than two statements in practice, the improvement in memory space of this method over standard paging is a factor proportional to the number of statements in the longest loop in a program. Note that in practice this may be comparable to the improvement obtained by standard paging over non-paged memory schemes, although I/O activities may increase in general.

In the past, a great deal of work has been done on the problem of extracting array operations from standard programs for purposes of compiling for high-speed array machines. Based on this, we have implemented a comprehensive FORTRAN program analyzer for speeding up FORTRAN programs, and have found that a very high percentage of FORTRAN loops can be broken into array expressions and linear recurrences by loop distribution (as in Program 2). An important key to our success has been in obtaining very accurate data dependence tests for subscripted variables inside loops. While a number of earlier attempts to solve this problem used only array names or simple subscript tests, we now use tests that in most cases are exact, i.e., we have necessary and sufficient tests for the data dependence of one subscripted variable on another, subject to a loop index set. This allows us to obtain a data dependence graph that has many fewer arcs (and more $\pi$-blocks) than would be obtained by more naive tests and, in particular, allows the breaking of many false cycles. Thus, control in most graphs can be distributed to the level of individual assignment statements.

An outline of the complete transformation algorithm is shown in Figure 1. The first step, analysis, is done automatically by the FORTRAN program analyzer. During this step, data dependences are determined, and certain simplifying transformations are performed (e.g., DO-loop initial value and bound normalization).

Following analysis, statements are clustered depending on common data elements. Consider, for example, Program 4. Statements $S_1$ and $S_2$ are clustered together because array $B$ is common to both statements. Clustering is done only within a loop however, so statement $S_4$ is not clustered with $S_3$ (yet) because they belong to different loops. Each cluster is called a name partition (NP). Loops are now distributed over NPs, as shown in Program 5. Notice that the loop could have been further distributed over the NP($S_1, S_2$). However, while this would reduce the space requirement for each loop, it would increase the page faulting since each page of $B$ would have to be fetched twice.

Following clustering, an attempt is made to fuse different loops together. Observe that the data in $S_4$ of Program 5 is a subset of the data of $S_5$. Thus, these two loops can be fused (as shown in Program 6) without increasing the space required in the loop, but allowing a decrease in the total number of page faults.

**Program 4**

```
DO 3 I=1, N
  S_1 A(I) = B(I) + C(I)
  S_2 D(I) = E(I) + F(I) + X(I)
  S_3 G(I) = B(I) + H(I)
  S_4 E(J) = D(J) + F(J)
```
arcs due to assignment statements to scalar variables. We use one of two techniques to handle such assignment statements—forward-substitution or scalar expansion. As an example, consider Program 7a.

**Program 7a**

```plaintext
DO S3 I=1, N
S1  T = (A(I)*C(I))/2
S2  D(I) = D(I)**2 - T**.5
S3  F(I) = T*(A(I)-C(I))/2 + F(I)/C(I)
```

Because of $S_1$, the data dependence graph of this NP is cyclic and there is one $\pi$-block. For this NP, the amount of memory allotment needed to obtain minimum I/O activity is four page frames. By substituting the right-hand side expression of $S_1$ in $S_2$ and $S_3$, we can eliminate $S_1$ and we will have two $\pi$-blocks: $\pi_1 = S_2$ and $\pi_2 = S_3$. Note that the memory needed for each of these $\pi$-blocks is three page frames. Thus the space requirement of Program 7a can be dropped by a factor of $\frac{2}{3}$ by forward-substitution and then distributing the control on the $\pi$-blocks as in Program 7b.

**Program 7b**

```plaintext
DO S2  I=1, N
S2  D(I) = D(I)**2 - (A(I)*C(I))/2**.5
DO S3  I=1, N
S3  F(I) = ((A(I)*C(I))/2)*(A(I)-C(I)) + F(I)/C(I)
```

In other situations, forward-substitution might be impossible or undesirable (e.g., if it increases the space requirement of the program). In such cases, we use scalar expansion as shown in Program 8.

**Program 8a**

```plaintext
DO S2  I=1, N
S1  T = T+A(I)*E(I)
S2  A(I) = B(I)*C(I)
S3  B(I) = T+F(I)*D(I)
```

Note that for Program 8a the memory requirement is six page frames while for Program 8b it is four (the maximum of the space requirement of the three $\pi$-blocks). Rules to be used in choosing between forward-substitution and scalar expansion (when both are possible) are discussed in Reference 3.

As mentioned earlier, distributing the control of an NP on its $\pi$-blocks will increase the page fault rate if the arrays referenced are multi-page arrays. To prevent this from happening, we apply the page indexing transformation to such loops. Program 8c shows the page-indexed distributed version of Program 8a. Basically, this transformation ensures that a page that is referenced in different $\pi$-blocks of an NP will not be removed from memory until it is used by all relevant $\pi$-blocks. (Additionally, by page-indexing we have reduced the size of the scalar expansion array in Program 8c to only 2 words instead of $N$.)

**Program 8b**

```plaintext
DO S2  I=1, N
S1  T(I) = T(I-1)+A(I)*E(I)
DO S3  I=1, N
S3  B(I) = T(I)+F(I)/D(I)
```

**Program 8c**

```plaintext
DO S2  I=1, N
S1  ILB = I+(IP-1)*Z
S2  A(I) = B(I)*C(I)
S3  B(I) = T(I)+F(I)/D(I)
```

Page indexing can be applied only to NPs which have basic $\pi$-blocks. A basic $\pi$-block is one with all of its statements at the same loop nesting depth. However, we have an algorithm for transforming non-basic $\pi$-blocks into basic $\pi$-blocks. After this is done, page-indexing is applied as before. Program 9a is a non-basic $\pi$-block (this is a Gaussian elimination program). In Program 9b, it is transformed to a basic $\pi$-block. Page-indexing is applied as shown in Program 9c. We use the same storage scheme for all multidimensional arrays of a program, the sub matrix storage scheme. This is because this storage scheme has inherent advantages over the row or column-wise storage schemes as was shown in 10.

We have developed tests to check the correctness of the

---

**Figure 1**—Flowchart of the transformation process.
page-indexing transformation. Currently, we are looking into storing arrays using different schemes when the arrays are referenced in different uniform ways in the program.

**Program 9a**

```plaintext
DO S2  I2=1, N-1
DO S1  I1=(I2+1), N
A(I2, I1) =A(I2, I1)/A(I1, I2)
```

**Program 9b**

```plaintext
DO S2  I2=1, N-1
DO S1  I1=(I2+1), N
A(I2, I1) =A(I2, I1)
```

**Program 9c** (Note we have substituted \( K \) for \( I_1 \), \( J \) for \( I_2 \). We assume \( RZ \) divides \( N \).

\[
RZ=Z^{**.5}
\]

```plaintext
DO S1  K=1, NP
KLB=1+(K-1)*RZ
DO S2  JP=KP, NP
JLB=1+(JP-1)*RZ
JUB=JP*RZ
DO S1  IP=KP, NP
ILB=1+(IP-1)*RZ
IUB=IP*RZ
IF (IP.EQ.KP) KUB=KP*RZ-1
IF (IP.NE.KP) KUB=KP*RZ
DO S2  K=KLB, KUB
IF (IP.EQ.KP) ILB=K+1
IF (IP.NE.KP) ILB=K+1
DO S2  J=JLB, JUB
IF (J.EQ.JLB.AND.IP.EQ.KP)
A(I, K)=A(I, K)/A(K, K)
DO S1  J=JLB, JUB
IF (J.EQ.JLB.AND.IP.EQ.KP)
A(I, K)=A(I, K)/A(K, K)
```

**PRELIMINARY RESULTS**

We define performance in terms of three criteria—space \((m)\), time (measured in terms of page faults \((f(m))\)), and space-time product \((f(m)m)\). Notice that for simplicity we assume that computation time is negligible relative to the time for a page fault, so we can measure time in terms of the number of page faults. In fact, our transformations do increase the CPU time for a program due to increased loop control overhead, redundancy (from forward-substitution of scalars), etc. However, much of this is invisible due to control-execution overlap, and the total increase in CPU time is generally negligible when compared to disk access time.

We define \( m_s \) to be the amount of memory required by a program in order to produce the minimum space-time product. More specifically, for the untransformed program this value is \( m_s \), and for the transformed program it is \( m_t \). Our choice of a memory allotment that minimizes space-time product is somewhat arbitrary, but is based on the intuitive idea that this will lead to maximum throughput and minimum turnaround in a multiprogrammed environment. In such a system, throughput and turnaround time are related by

**Throughput = average number of jobs present / average job turnaround time**

Roughly speaking, reducing page allotments as much as possible maximizes the average number of jobs present. Since reducing each job's page faults reduces the average job turnaround time, the transformations we carry out tend also to maximize throughput. Thus, for the fixed memory allotment case discussed above, our techniques improve both turnaround and throughput. In fact, however, \( m_t \) does not always lead to minimum page faulting as we shall see. Thus, in an I/O bound system, \( m_t \) may not produce optimal results. We shall discuss this problem in more detail later.

Regarding space, it is intuitively clear that if one can transform a program into a form that contains a set of \( \pi \)-block computations, then each of these \( \pi \)-block computations can be broken into a sequence of page-sized loops (using the page indexing transformation). The net effect would be to reduce the necessary page allotment for an entire transformed program to that required for the largest \( \pi \)-block computation in the program, \( m_t \). The simplest \( \pi \)-blocks might be expected to contain one or two distinct arrays, while the maximum number per \( \pi \)-block in a program might be six or eight. Thus if one were successful, perhaps any program could be run with a data page allotment of at most eight pages.

Another important program characteristic is page faulting and how it would be affected by such transformations. For any given program loop, the number of page faults per iteration equals the number of arrays referenced in the loop, unless all pages are left in main memory for the entire loop execution, in which case page faulting occurs only after a number of iterations proportional to the page size. In this case the page allotment for the original program, \( m_s \), would at best be equal to the largest number of arrays referenced in any loop in the program, in contrast to the \( m_t \); the largest \( \pi \)-block obtained above. At worst, many more pages than array names would be required. For example, a matrix multiplication loop has three array names but needs an entire row and column of pages. Thus, one would expect that the number of required pages for any program to run well would be less for a transformed program, i.e., \( m_t < m_s \).
By a combination of automatic and manual transformations, as well as a tracing program that handles FORTRAN programs compiled for the IBM/360, 370, we have obtained preliminary statistics for 17 FORTRAN programs, comprising a total of almost 1600 cards (excluding comments). These were selected from about 300 programs we have collected for various studies. The 17 programs contained a total of 200 DO-loops whose limits were supplied by users. The program generated over 1.4 million array element references. Most of the programs are numerical in nature but they are drawn from diverse application areas. One important criterion in our selection process was that the programs not contain too many statements, as our analysis procedures were rather time-consuming.

We have observed average values of $m_a=25$ and $m_t=4$. Thus, a space-saving of greater than six could be expected for these programs. However, since page faulting does increase somewhat for the transformed programs, the space-time product improvement is only about three or four (the mean is four and the median is three). Nevertheless, this implies a potential increase in throughput of a factor of three or four for a multiprogrammed system. Furthermore, for our programs the average space-time product improvement of standard paging over a non-paged main memory (one that allocates sufficient space for all arrays) was only a factor of about 2.5 (mean and median). Thus, the performance increase of our transformations over standard paging is comparable to (in fact, greater than) the performance increase of standard paging over a non-paged system.

It is important to realize that our $m_a$ values were obtained by direct observation of the space-time product of our sample programs over a wide range of memory allotments. The difficulty of directly observing $m_a$ (say, by compiler measurement) has been discussed in Reference 20, where possible correlation with the number of array names in a loop was rejected as impossible in practice. On the other hand, for our transformed programs there is a strong correlation between the number of array names and $m_t$; the two are almost equal in most $t$-blocks.

In Figure 2 and Figure 3 we show typical page fault versus memory allotment and space-time product versus memory allotment curves, respectively. Note that both page fault curves approach approximately the same low level of page faulting. The transformed program approaches this low level much earlier than the original program. This, in turn, causes the space-time product of the transformed program to be substantially less than that of the original program over a wide range of memory allotments. Note that the transformed program's space-time product decreases monotonically to a minimum at $m_a$, then increases to meet the minimum for the original program at $m_t$, and then both increase together as useless memory is allotted. However, the space-time products of original programs usually vary a great deal between one page and $m_t$, thus making the operating system's (OS) page allotment job nearly impossible unless $m_a$ pages are allotted (the difficulty of knowing $m_a$ was discussed above). Of course, allotting each program an efficient amount of space would also complicate the overall scheduling job of the OS. However, the transformed programs are much better behaved, and any allotment in the region of $m_t$, say from four to eight pages will give a reasonably good space-time product for any of the programs we observed.

To compare averages using ideal memory allotments ($m_a$ and $m_t$) is rather pointless because of the difficulty of achieving $m_a$ allotments in practice. More realistic is a comparison of the performance of the original and transformed programs given fixed allotments, $m_a$, which are not necessarily optimal. This reflects the situation where space is allotted by an OS that does not know the optimal allocation. (Note, however, that it may be easier for a compiler to estimate $m_t$ even though estimation of $m_a$ is known to be difficult.) We have tabulated the ratio $m_a f_a(m_a)/m_t f_t(6)$, for $16 \leq m_a \leq 48$ with $m_a$ increasing by increments of four. For $m_a=16$ the space-time product of transformed programs is improved over untransformed programs for all but one of the 17 programs (where it drops by a factor of $.75$), with an average improvement by a factor of 8.8. Further
thermore, in all but three cases, both space and page faults decrease (in two of these the page faults increase very slightly). As \( m_n \) increases, several other programs reach \( m_n \) and need no more pages, but averaging over those that need 28 pages (14 programs), the average space-time product improvement reaches a maximum of 12.9 with a median of seven. In fact, over the entire range of \( m_n \) and using four, six, or eight pages for the transformed programs, we achieve an average space-time product improvement over untransformed programs in the range of seven to over 12. Thus, we feel safe in concluding that by using our transformations, operating systems that use fixed memory allotments can achieve a decimal order of magnitude improvement in space-time product over standard paging techniques for the data of ordinary FORTRAN programs.

The above studies assumed a fixed memory allocation and a least-recently-used (LRU) page replacement algorithm. We have done similar studies assuming paging is done according to the working set policy, WS. For transformed programs, there was no difference between the cost of execution under LRU and WS. Moreover, several programs exhibited the working set anomalies. For more discussion on the results under WS, see References 3 and 1. For detailed measurements of the working set anomalies in FORTRAN programs, see Reference 2.

CONCLUSION

In this paper we presented an overview of compiler transformations which are aimed at the enhancement of the locality property of programs. Moreover, we presented a summary of preliminary experimental results which show that our techniques have good potential for achieving their goals. These results indicate that transformed programs are cheaper to execute, easier to manage, and simpler to model. For example, using simple and practical memory management policies, we observe a factor of 10 improvement in space-time cost over untransformed programs.

REFERENCES