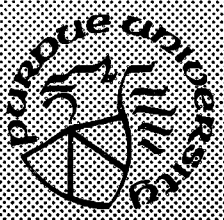
4-1-1990

Data Layout Optimization and Code Transformation for Paged Memory Systems

Lung-Yu Chang
Purdue University

Henry G. Dietz
Purdue University

FILE COPY



Data Layout Optimization and Code Transformation for Paged Memory Systems

Lung-Yu Chang
Henry G. Dietz

TR-EE 90-43
April 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Data Layout Optimization and Code Transformation for Paged Memory Systems

Lung-Yu Chang and Henry G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
April 1990

ABSTRACT

Supercomputers need not only to have fast functional units, but also to have rapid access to massive quantities of data. Virtual memory paging and physically distributed memory systems both attempt to provide this large data space, but performance of a computer system using either memory organization is highly dependent on the page reference pattern and the number of pages available locally. Despite this, surprisingly little work has been done toward using the compiler to optimize memory system performance. In this paper, we introduce compiler techniques which use a combination of *data layout* and *code transformation* to improve paging performance for compiled programs. These same techniques can also be applied manually to improve performance using existing compilers.

Keywords: compilers, paging, data layout, code transformation.

1. Introduction

Especially given recent advances in VLSI technology, it is relatively easy to construct high speed computing elements. However, to achieve good system performance, faster functional units require faster access to more data. Whereas traditional compiler optimization is primarily concerned with optimizing operations, this recent trend leads naturally to the concept of optimizing data accessibility — which can be achieved partly by changing the data layout and partly by changing the order in which references are made (using code transformations). Although this concept applies to both serial and parallel machines, this paper focuses on paged memory systems in serial machines.

There are three fundamentally different approaches to using the compiler to improve paging performance:

[1] *Data layout* — the reference pattern is changed by changing the layout of the data structures in memory space, or perhaps even changing the logical data structures

[2] *Code transformations* — the reference pattern is changed by transforming the code which makes the references so that the references are made in a more desirable order

[3] *Control directives* — the runtime environment is given static information about the program's reference pattern which enables it to adjust the memory management scheme for better performance

This paper discusses a number of techniques for [1] and [2], a few of which were suggested in [ChD89]. (Note that any techniques for [1] and [2] can not only be applied by the compiler, but also by motivated programmers using existing compilers.) Although we feel that [3] is also important, it is not a major concern of this paper.

Section 2 summarizes previous work toward improving paging performance using program optimizations. In section 3, compiler optimization of data layout is discussed. Code transformation techniques are presented in section 4. Finally, conclusions are given in section 5.

2. Background

Although there has been very little done toward compilers automatically optimizing data layout and performing code transformations to improve paging performance, a number of studies have examined the issues involved in manually applying these concepts.

In [McK69] different storage patterns (row-major, column-major, and block-oriented) were investigated to determine which yielded minimum page faults. In order to increase locality of the program under different storage patterns, strip-mining and loop reversal techniques were employed and hand-coded by the authors. Given that a single page is not large enough to hold an entire array, closed-form expressions to compute page-fault counts were derived for a set of linear algebra algorithms; the block-oriented storage pattern was most efficient.

However, this assumed that *all* arrays would be stored using the same storage pattern — the possibility of allowing each array to have its own storage pattern was not considered. Further, it

is not clear that the block-oriented storage pattern is generally preferable, since only a few numerical algorithms were examined.

The notion that each data structure can have its own storage/access pattern can be found in [Leu81], [Mac85], [Mac88]. In [Leu81] Leuze considered memory access patterns for some commonly used high level operation (e.g. matrix-vector multiplication) for vector computers. By relaxing the way vectors can be referenced from the memory many algorithms that contain short vector operations can be vectorized to contain fewer and longer vector operations, thus increasing the vector machine performance. He also discussed hardware implementation for generating addresses for such access patterns. Timing statistics were computed for different access patterns for each high level operation and the optimal one was chosen. Although this is useful when applied to frequently used high level operations, such as matrix algebra, this method can hardly be automated since the access patterns are determined by the particular high level operations.

In [Mac85] [Mac88], Mace considered the problem for the whole program represented as a DAG (directed acyclic graph) where each node denotes a high level operation. This technique works, but is limited to a vector model of computation: the high level operations can be most conveniently expressed in vector languages and it is relatively easy to compute the accessing cost (depending on different strides associated with different accessing/storing pattern) for vector computers.

The idea of juxtaposing elements of a data structure with elements of another data structure according to the access pattern has appeared in the guise of layout of data bases [Mar75]. In physical data base design, in order to improve locality in accessing records sequentially, one would place records containing the various fields in an array of such records instead of implementing each field as a separate array (as is traditionally done in languages such as FORTRAN).

In addition to data locality, program code locality has also been investigated. Hatfield [HaG71] considered the problem of how to layout the program code sectors (blocks) such that the page faults can be minimized when the program is executed, assuming that sizes of program sectors are smaller than page size. Some heuristics were proposed based on the traffic among sectors. Lau and Ferrari [LaF83] adopted a similar approach. They consider both program code and data locality and employ simple heuristics to determine layout of program code and data in the virtual address space. However, since conventional data layout was assumed, the extent of data restructuring is limited to layout of rows of arrays relative to each other.

Rather than attempting to improve the match between access pattern and storage pattern by changing storage pattern, it has been far more common to investigate rewriting the code to effect a different access pattern. Moler [Mol72] suggested applying loop interchange in some basic and frequently used subroutines to improve performance. Guertin [Gue72] also discussed loop interchange and further suggested that rearranging terms within expressions may also be helpful. In addition to loop interchange, Elshoff [Els74] put forth techniques amounting to what are now known as strip-mining and loop reversal transformations. Trivedi [Tri77] suggested automating

loop reversal and loop distribution and gave conditions for doing so; he also suggested that programmers ought to write their code in terms of subarrays (implying a block-oriented storage layout). Abu-Sufah [Abu79][Abu81] added techniques including statement clustering for loop fission and fusion, the experiments he conducted showed that memory requirement and the page faults could be improved by an order of magnitude.

In order to using program structure information to aid memory management decision, a compiler-directed memory management policy has been proposed [MAP85][MaP88]. Impressive performance has been observed, however, the overhead on the operating system to implement this method is high because the operating system must track how many pages are in the memory and which arrays these pages correspond to for each process at every stage of execution.

3. Compiler Optimization of Data Layout

We define an optimal data layout for a program to be a mapping of data elements into memory locations such that the minimum total execution time is obtained for the program. Without presenting a formal proof, it is easily seen that determining the optimal data layout is an NP-complete problem. Informally, if there are *n* data, then there are at least *n!* possible data layouts to be examined (more if one allows "gaps" in memory) and even if each could be examined in constant time, O(*n!*) complexity would still mean that the problem is NP-complete.

Hence, there are two basic approaches. The first, which is discussed in section 3.1, is to simply require the user to explicitly specify the data layout to be employed for each data structure. The second, which we feel is much preferred, is to allow the compiler to determine a good layout for each data structure by using suboptimal data layout transformations. Section 3.2 discusses the heuristic algorithms which we propose for compiler optimization of data layout.

3.1. Explicit Data Layout

It is surprising that most modern high-level computer languages do not separate the concept of a data structure from the memory layout of that structure. In other words, when the user specifies a data structure, the specification implies a particular data layout. For example, in FORTRAN, all two-dimensional arrays are *required* to be laid out in a *column-major*[1] fashion. In PASCAL and C, it is *required* that such arrays be represented row-major in memory. It is further specified in most modern HLLs (high-level languages) that each array has its own designated

1. There are many variations on this terminology: *column-major* is also called *column-wise*, *column order*, etc. It means simply that the i,j^{th} element of a two-dimensional array is located at an address computed by a formula like: *array base address + (i * size of one element) + (j * size of one element * number of values for first subscript)*. If instead the layout addressing were computed by: *array base address + (j * size of one element) + (i * size of one element * number of values for second subscript)*, then this would be called *row-major*.

storage area; if there are two arrays, `a` and `b`, the elements of `a` are defined as being an uninterrupted sequence in memory — it is not permitted that elements of `b` would be interspersed with those of `a`.

The fact that data structure *implies* data layout simplifies the task of a dumb compiler in that it permits the compiler to blindly map data structures to memory locations using a few simple rules. However, these rules are not sensitive to how the data structures are accessed in the program; hence, the layout is efficient if the programmer is *very careful*, but otherwise the layout is likely to be quite inefficient. For example, declaring:

```
int a, b, c, d;
```

where two integers "fit" in a page would probably place `a` and `b` in one page and `c` and `d` in another. Yet, if `a` and `d` are always referenced together and `b` and `c` are always referenced together, the system would require two pages for each set of references — placing `b` and `c` on one page and `a and d` on another would reduce this to using just a single page for each set of references. Obviously, if only one page is likely to be available, the second layout is far more effective.

Unfortunately, few applications programmers would be aware of, and willing to deal with, such issues. However, if the user is willing, the "compiler techniques" described in the rest of this paper also serve as guidelines in manually improving performance under existing compilers.

3.2. Compiler Optimization of Data Layout

Because existing languages imply data layout in their data structure specifications, it is fairly difficult for a compiler to determine when changing the data layout can be accomplished without changing the functionality of the program. We view this issue as primarily one of language design rather than compiler analysis; slight changes in language semantics easily remedy the problem. For example, members of each C `struct` are required to layout in the sequence of definition, but few programs would "break" if the semantics were changed to allow the compiler to reorder the members.

The most obvious advantage in having the compiler determine data layout is that it frees the application programmer from that burden. It is also likely that the compiler, despite using suboptimal heuristic algorithms for data layout, will generally outperform programmers because the analysis is too tedious for most programmers to bother with. In fact, many high-level languages (HLLs) lack the constructs needed to express the optimal data layout — for example, Fortran 77 does not provide a way to interleave arrays (in C this can be accomplished by defining an array of a `struct` type). However, the greatest advantage has to do with the relationship between data layout and final code structure.

In general, data layout is orthogonal to code optimization — i.e., one can change either without altering the other, although performance may change dramatically. Although a user might be able to optimize data layout for the code *as he wrote it*, that data layout might be very poor for the code structure as the compiler generates it. Since only the compiler "knows" what code transformations will occur, only the compiler is able to adjust the data layout to better match the final code structure. Further, because the two aspects are orthogonal, the compiler can literally transform the code and then adjust the data layout to match — a very simple procedure.

Of course, because finding the optimal data layout is NP-complete, the compiler algorithms used will be suboptimal and of limited generality. The main restriction that we impose on programs is that they operate only on scalars or array elements. Further, we assume that each array is either one or two dimensional and is larger than the size of one page. These restrictions are met by most scientific applications codes. Sections 3.2.1-3.2.4 describe four heuristic algorithms for compiler optimization of data layout.

3.2.1. Packing Scalar Variables

If there are relatively few scalar variables, very good performance can be achieved by simply packing them all into a single page. However, if this is not the case, techniques proposed in [HaG71], [MSN74], [Fer74], [Fer76] can be used to pack the scalars into the pages. A simple heuristic such as the following may suffice: in scanning the intermediate code, as each variable is referenced for the first time, pack it into the next available location.

3.2.2. Data Replication and Renaming

Renaming is a technique used to get rid of spurious dependences — *i.e.* anti and output dependences[KKL81] (or called storage related dependences [CyF87]) such that more parallelism can be uncovered. On the other hand, renaming can be useful in allocating storage [CyF87][Fab79][CLZ86] such that minimum storage for program execution can be achieved.

Renaming is also effective in reducing page faults just as in uncovering parallelism and minimizing storage requirement — by renaming, one can lift the constraints due to sharing of storage that is specified in the original program. In view of paging performance, renaming allow us to optimally match the storage patterns with the access patterns, which is impossible without renaming since different access patterns to the same array can coexist in the original program.

Scalars not in a loop can be renamed with traditional compiler techniques [AhU77][CLZ86][CyF87]. An algorithm was developed to rename scalars in the loop [KKL81]. However, their algorithm can not be applied for array renaming since statement as the granularity of dependence is too coarse to be adequate, one must view each reference of array as the granularity of dependence. For arrays to be renamed, this has been mentioned in [Die87][CyF87]; the treatment of array is not different from that of scalars except that when only portion of the arrays get updated, extra copy instructions are introduced to copy all the renamed array elements back to

the original array. In the following, we will give some sufficient conditions for renaming arrays referenced in the loops, this also include renaming of portion of the arrays. Before we do that, some definitions are in order.

[Definition]

A reference α of an array A is *universally dependent* on another reference β of A if all array elements accessed by α have been accessed before by β. This definition can be generalized when α has more than one references on which it depends.

Depending on what kind of references to the array, one can similarly define universal flow, anti, and output dependences. It is clear that renaming can be performed when there is a universal anti or output dependence. However, for a universal flow dependence, one can still perform renaming by adding extra copy instructions; this can be beneficial when access patterns of the two references are different. Therefore we can state the condition of renaming to be: Perform renaming if there exist universal dependence between two references that belong to two basic blocks of different loops and the access patterns of the references are different from each other, or if there exist anti or output universal dependence between these two references.

To make the renaming condition more general, we define dependence graph of array references G(N,E), where N is the set of nodes denoting references and E is the set of edges denoting dependences; each edge is represented by an index set to denote the domain of dependence between the two nodes, which is a region in the array that causes the dependence. Note that there may exist a cycle between two nodes but in this case the two arcs must be denoted by two disjoint domains. This can be seen from the following code:

```
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    A[i][j] = ...
    A[j][i] = ...
  }
}
```

For the domain B = {(i,j) | 0≤i≤N-1, 0≤j≤N-1, i≤j}, A[j][i] depend on A[i][j], and for the complement of B, A[i][j] depends on A[j][i]. Note that arcs made redundant by transitive closure do not appear in the graph; if there is an arc from node n_i to node n_j and an arc from node n_j to node n_k, then there is no arc from node n_i to node n_k. The dependence graph can be constructed by dependence analysis [Kuh80][KKL81].

The renaming condition can be made more general: If there are more than one access patterns of references to the same array, perform renaming to a reference if this reference is involved in no cycle of dependence and all its predecessors in the dependence graph have different access patterns; renaming can also be performed if universal anti or output dependences exist between a

reference and all its predecessors. The following algorithm is used to rename array references, given graph G(N,E).

[1] Find out all the cycles and the associated nodes in the graph, mark all the nodes in cycles.

[2] Perform topological sort on the remaining unmarked nodes of the graph (it is now acyclic), renaming the nodes accordingly based on the above conditions. Note that extra copy instructions may be necessary when involved dependence is a flow dependence.

An example to show how this technique is useful is to compute the square of a matrix A as follows:

```
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    A[i][j] = ...
  }
}
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    for (k=0; k<=N-1; k++) {
      B[i][j] = B[i][j] + A[i][k] * A[k][j];
    }
  }
}
```

Since the second reference of array A in the second loop has access pattern different from that of the reference when it last get updated, one can replicate array A into another array C and rewrite the original program as follows:

```
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    A[i][j] = ...
  }
}
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    C[j][i] = A[i][j];
  }
}
for (i=0; i<=N-1; i++) {
  for (j=0; j<=N-1; j++) {
    for (k=0; k<=N-1; k++) {
      B[i][j] = B[i][j] + A[i][k] * C[k][j];
    }
  }
}
```

If we assume that at least three page frames are allotted for execution, then it is clear that number of the page fault of the original program for any storage pattern is larger than that when array A,B are row-major stored and C is column-major stored after replicating.

An extension to the methods proposed by [Die87][CyF87] is that when portion of an array gets updated, it can be renamed without copying back to the original array if later references of the array in the program do not access the array accrossing the boundary of the portion.

3.2.3. Array Element Layout Pattern

Using our compiler-directed memory management scheme, a certain number of page frames is allocated to the process before execution of each phase. With the allocated page frames for each phase, we are able to determine the possibly best storage pattern for each individual array, based on its access patterns in the program.

In order to determine the storage pattern of each array, we make the following assumptions in each phase of the program:

[1] There is a separate storage region for each array.

[2] Each phase is a doubly nested loop with known reference patterns for each array, the loop is normalized such that for each level of the loop the lower bounds of the index variables are zero and the increments are one. Let the upper bound of outer and inner loops are M-1 and N-1, respectively.

[3] Let the fixed page size P and all loop upper bounds be power of 2.

[4] Fixed number of page frames are allocated.

[5] For a two-dimensional array A assume its reference pattern is

$$A\left(i\begin{bmatrix}a\\b\end{bmatrix}+j\begin{bmatrix}c\\d\end{bmatrix}+\begin{bmatrix}e\\f\end{bmatrix}\right)$$

where i is the index variable of outer loop and j is the index variable of inner loop. Written in C, the above reference is expressed as A[a*i+c*j+e][b*i+d*j+f].

The following steps are used to estimate the number of the page faults for each array A in each phase with block-oriented storage pattern p_1 by p_2.

[1] Classify all the references of A as follows:

$$A\left(i\begin{bmatrix}a_1\\b_1\end{bmatrix}+j\begin{bmatrix}c_1\\d_1\end{bmatrix}+\begin{bmatrix}e_1\\f_1\end{bmatrix}\right)$$

and

$$A\left(i\begin{bmatrix}a_2\\b_2\end{bmatrix}+j\begin{bmatrix}c_2\\d_2\end{bmatrix}+\begin{bmatrix}e_2\\f_2\end{bmatrix}\right)$$

are in the same class if $a_1 = a_2$, $b_1 = b_2$, $c_1 = c_2$, $d_1 = d_2$, and also

$$\left|e_1 - e_2\right| \le \frac{p_1}{2} \quad \text{and} \quad \left|f_1 - f_2\right| \le \frac{p_2}{2}$$

[2] For each class, the estimate of number of page faults is

$$M\frac{N}{\min\left(\left\lfloor\frac{p_1}{|c_1|}\right\rfloor,\left\lfloor\frac{p_2}{|d_1|}\right\rfloor\right)}$$

We note that this formula is a very rough estimate (it underestimates within factor of 2 for all reference patterns), but it does indicate how well the chosen storage fit the array reference pattern.

[3] Summing over all classes of array A we can get an estimate of page faults of array A in this phase. Therefore we can get an estimate of total page faults of array A in the whole program by summing estimated page faults over all phases.

Note that page faults for one dimensional arrays can be easily calculated and when the loop is not perfectly doubly nested it can also easily be calculated based on the above formula. The "optimal" storage pattern of array A can be obtained by varying the sizes of p_1 and p_2 under the constraints that their product is equal to page size P and both are power of 2, and choose p_1 and p_2 that yields the minimal estimate of page faults.

3.2.4. Juxtaposing and Overlaying Arrays

Up until now we assume that each array has its own designated storage area. However, if we strictly follow the principle that the optimal storage scheme should be such that access patterns match the storage scheme as close as possible, then there is no reason why one should not juxtapose the arrays in the data layout. For example, in the following codes:

```
for (i=0; i<=N-1; i++) {
  A[i] = B[i] + C[i];
}
```

If array elements of A, B, and C can be interleaved element-wise, the access pattern matches the storage scheme perfectly and consequently one need only one page frame in order to be able to execute the codes without many interruptions, one need three page frames to execute the program for the same number of page faults with the old data layout.

Another technique to reduce storagement requirement and number of page faults is to overlay data structures without violating program semantics. For scalar overlaying in register allocation different techniques have been proposed [Fre74] [Cha81] [ChH84] [CyF87] [Chi89]. However, these techniques can not be directly applied since there are two aspects we should consider in array storage allocation for a paged system with dynamic memory allocation policy as compared to scalar storage allocation in register allocation:

[1] A page frame can accommodate more than two array variables even if the page size is smaller than the array size and there exists interferences of the live ranges of the arrays due to the possibility of juxtaposition.

[2] Under the assumption of dynamic memory management, the number of page frames assigned to a program can vary depending on the access pattern in each program phase. Although there are many page frames in main memory that can be assigned to each individual process, in a multiprogramming system it is unlikely to make so many page frames available for a single process; on the other hand, there is no predetermined small number of storage units that can be assigned to each process as in register allocation, thus each process has some degree of freedom for more storage. Therefore it remains to choose proper and limited number of page frames to be assigned for program execution in each phase without excessive amount of page faults.

Clearly the array allocation problem in the discussion is more difficult than the conventional register allocation problem. As a result, a heuristic approach is used to solve this problem. In order to do so, we first define what an overlay graph is.

[Definition]

An *overlay graph* G(N,E) is a directed graph where N is the set of nodes containing live ranges of arrays, and E is the set of edges such that if $(n_1, n_2) \varepsilon E$, then n_1 can be overlaid with n_2 and there is no other live range n_3 such that n_1 can be overlaid with n_3 and n_3 can be overlaid with n_2.

In other words, $(n_1, n_2) \varepsilon E$ means that storage assigned to n_1 can be assigned to n_2 and if another live range n_3 can be assigned the same storage, then n_3 can not be between n_1 and n_2 in execution ordering. In essence, the overlay graph is complement of interference graph mentioned in register allocation algorithm except that it is directed and redundant edges are removed such that if there is a path of length greater than one between two nodes n_1 and n_2, then there is no edge in G between these two nodes.

The reason that overlay graph instead of interference graph is used in our array allocation algorithm is that it contains more flow specific information and allow us to interleave arrays when appropriate. The array allocation algorithm, therefore, consists of two subalgorithms, one is overlay graph construction which in turn utilize node insertion algorithm and the other is coloring.

Algorithm Node_Insertion(G,α)

INPUT:

An overlay graph G = (N,E) and an array live range α.

OUTPUT:

A new overlay graph containing α and the nodes in G.

METHOD:

[1] If G is an empty graph, let G be the graph containing only node α, return G.

[2] For convenience, denote $\alpha < \beta$ if live range α can be overlaid with β (*i.e.* no overlap of these two live ranges and references of α precede references of β elementwise). For each node β in G, if $\alpha < \beta$ or $\beta < \alpha$ then add node α and an edge (α, β) or (β, α), respectively to G.

[3] For each original edge (β, γ) in G,

[3.1] If (β, α) and (α, γ) belong to the new graph, delete (β, γ) from the new graph.

[3.2] If (β, α) and (γ, α) belong to the new graph, delete (β, α) from the new graph.

[3.3] If (α, β) and (α, γ) belong to the new graph, delete (α, γ) from the new graph.

TIME COMPLEXITY:

O(|N|+|E|).

Algorithm Graph_Construction(S)

INPUT:

A set S of array live ranges.

OUTPUT:

The overlay graph for these array live ranges.

METHODS:

[1] Let G be an empty graph.

[2] Pick a live range α in S, if G is empty, let G be the graph containing only node α; otherwise apply the above node insertion procedure to insert α to G. Set S = S - {α}.

[3] If S is empty, stop, else goto [2].

TIME COMPLEXITY:

The node insertion is $O(|N_i|+|E_i|)$, each $|E_i|$ is at most $O(i^2)$, hence total complexity is $O(|S|^3)$.

Algorithm Coloring(G)

INPUT:

An overlay graph G.

OUTPUT:

A coloring of the graph.

METHOD:

[1] Initially all nodes are not colored and marked.

[2] If there is no unmarked node then stop; else choose an unmarked node n that has no unmarked ancestor. If n is colored, mark n and go to [2]. If all n's immediate ancestors have the same color, color n with that color. Mark n and go to [2]. If n has no uncolored child, mark and color n with a new color; go to [2].

[3] Otherwise assign a new color to n if n is not colored yet, choose those n's uncolored children which have the same live range as n and juxtapose them and then overlaid with n; Mark n and go to [2].

[4] Otherwise find an uncolored child m of n such that m is the only uncolored child of m's other immediate uncolored ancestors. Let set T contain the live ranges of m's other ancestors that have the same live range as n does. Assign a new color to n if n is not colored yet and node m and all live ranges in T get the same color as n does by juxtaposing all live ranges in T and n and then overlaying with m; Mark n and go to [2].

[5] Otherwise choose an uncolored child m of n which has minimum number of immediate uncolored ancestors, if n is colored then color m with the same color else assign a new color to n and m by overlaying m with n. In case of tie, choose m to have minimum number of immediate ancestors. Mark n and go to [2].

TIME COMPLEXITY:

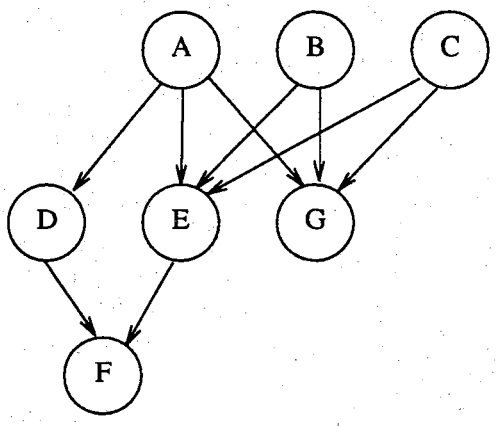
Step [1] is O(|N|), [2] is O(d)| d is the maximum degree for any node, [3] is $O(d^2)$, [4] is O(d), and [5] is O(d). Hence, total complexity is $O(|N|d^2)$.

Implicitly stated in the coloring algorithm is the juxtaposition pattern, this can be easily determined by accumulatively adding up the live ranges that are juxtaposed in the algorithm for the same color. After coloring, the compiler can compute the number of page frames required for each phase of program execution by counting different colors that are used to color the live ranges and pass this information to the operating system by generating directives. Note that although a phase in our definition is a rather fixed program region, the determination of number of colors to be used works for any region of program, one can simply count the colors used for the live ranges in that region.

The following example shows how the coloring algorithm works:

```
for (i=1; i<=N; i++) {
    a[i] = b[i] + c[i];
    }
for (i=1; i<=N; i++) {
    d[i] = a[i-1] * b[i];
    e[i] = b[i] - c[i] + d[i];
    }
for (i=1; i<=N; i++) {
    g[i] = e[i] + d[i];
    }
for (i=1; i<=N; i++) {
    f[i] = e[i] - d[i];
    }
```

The overlay graph for this algorithm is



To color this overlay graph, first choose node A, since its children D and E have the same live range, we color A,D and E with color 1 according to step [3] by first interleaving D and E and then overlay A. Mark A. Next we choose node B, which has node G as the only uncolored child that is the only child of G's ancestor(s) other than B (*i.e. C*), since C has the same live range as B, according to step [4], one can color B,C, and G with color 2 by interleaving B,C and then overlay with G. We then mark B. With the nodes C, D, E, and G, they are marked according to step [2]. Finally for node F, since all its immediate ancestors D and E have the same color 1, we also color it with color 1 according to step [2]. Thus in total we need two colors to color the graph, consequently each phase in the program needs two page frames to be allocated.

4. Code Transformation Techniques

Although there are many conventional optimizations which may be viewed as improving paging performance (e.g., strip mining [Abu78]), we will discuss only the two new techniques which we have developed for this purpose. The first technique, statement alignment, restructures loops so that sequential array references cross each page boundary only once; this is described in

section 4.1. In section 4.2, a technique which buffers values between loop iterations in registers (thereby avoiding additional page references), is described.

4.1. Statement Alignment

The first technique is statement alignment [Wol82][ACK87], this technique has been applied in parallelizing loop execution. Consider the following loop:

```
for (i=1; i<=N; i++) {
  a[i] = b[i] + c[i];
  d[i] = a[i-1] * e[i];
}
```

It seems that only five page frames are required in executing this loop, assuming each array is allocated dedicated continuous region; however, when the value `i` is such that `a[i]` is at the beginning of a new page, two pages is then necessary for array `a` for execution. Thus six page frames are required in total. After alignment for array `a` we have the following loop:

```
for (i=0; i<=N; i++) {
  if (i>0) a[i] = b[i] + c[i];
  if (i<N) d[i+1] = a[i] * e[i+1];
}
```

Only five page frames are needed for arrays `a,b,c,d,e` for each iteration of the loop as the result of alignment. In the following we present an algorithm for statement alignment for the purpose of minimizing page requirements.

Statement Alignment Algorithm

[1] Construct dependence graph in which nodes represent statements and the arcs are labeled by array name, dependence vector, and dependence type, this is an extended model of [KKL81] in that array name and dependence type along with the dependence vector are associated with each arc. Initially all nodes are unmarked.

[2] For each node we define the up-shift vector as the lexicographical minimum of all distance vectors associated with the incoming arcs, the down-shift vector as the minimum of all distance vectors associated with the outgoing arcs. Note that arcs with input dependence type do not count as dependences in computation of up-shift and down-shift vectors.

[3] For each node compute number of additional alignment of shifting up and down by the amount of up-shift and down-shift vectors. This is the number of array references

1. Here dependence type also consists of input dependence in addition to flow dependence, output dependence, and anti-dependence.

associated with arcs with flow, output and anti-dependences getting aligned minus the number of the references associated with arcs with input dependences getting unaligned as a result of shifting. Take the maximum of positive additional shifting-up and shifting-down alignments as the number of possible alignment for each node.

[4] We then sort the unmarked nodes according to the number of possible alignment. Pick the node with the maximal number of alignment, if it results from shifting up, shift up all the references of the corresponding statement, otherwise shift down the references. Rewrite the loop with new appropriate lower and upper bound and the statements such that computation remain unchanged. Note that as a result of alignment, statement ordering might have to be changed because of dependence relation. Modify the dependence vectors of the arcs associated with the chosen node by the amount of shifting. Accordingly modify the up-shift and down-shift vectors and the additional alignments of the affected nodes as a result of dependence vector changes of the above arcs. Mark the node with maximal number of possible and the aligned nodes as a result of alignment.

[5] If all up-shift and down-shift vectors of all unmarked nodes (statements) are zero vector for all unmarked statements or all additional alignments are negative, stop; otherwise update possible alignments of affected nodes and go to step 4.

Clearly the time complexity of this algorithm is O(NE) where N is the number of nodes and E is the number of edges in the dependence graph.

4.2. Inter-Iteration Buffering

Another technique to reduce memory storage requirement is buffering. The basic idea of buffering is to store computation results in temporary storage space of higher memory hierarchy and store it back to memory space of lower hierarchy in later iteration of loops. This technique is relatively straightforward applied for one-dimensional arrays in a single loop. When two-dimensional arrays are considered, the following assumptions are made in order to apply buffering technique.

[1] Assume only one class of reference pattern for each array A and the reference pattern can be written as

$$A\left(i\begin{bmatrix}a\\b\end{bmatrix}+j\begin{bmatrix}c\\d\end{bmatrix}+\begin{bmatrix}e\\f\end{bmatrix}\right)$$

where a,b,c,d are fixed but e and f may vary for different references. Without loss of generality, let c and d be greater than zero.

[2] Only true dependences exist between two references. That is to say, there is only one write reference and the rest of references are read references. Further, the dependence distances are multiples of $\begin{bmatrix}c\\d\end{bmatrix}$.

[3] Each array starts with page boundary in virtual memory space. The storage pattern of the array A is block-oriented with size p_1 by p_2.

The method to apply buffering technique is simple: take the write reference and the closest read reference and test if the write reference is crossing page boundary, if so, buffer the computation result in a scalar variable.

For example, the C code:

```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    A[i+j+2][2*j+2] = A[i+j+1][2*j] + B[i][j];
  }
}
```

embodies a reference pattern which can be expressed as:

```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
```

$$A\left(i\begin{bmatrix}1\\0\end{bmatrix}+j\begin{bmatrix}1\\2\end{bmatrix}+\begin{bmatrix}2\\2\end{bmatrix}\right)=A\left(i\begin{bmatrix}1\\0\end{bmatrix}+j\begin{bmatrix}1\\2\end{bmatrix}+\begin{bmatrix}1\\0\end{bmatrix}\right)+B\left(i\begin{bmatrix}1\\0\end{bmatrix}+j\begin{bmatrix}0\\1\end{bmatrix}\right)$$

```
  }
}
```

Hence, the original C code can be transformed into:

```
flag = 0;
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    /* test if write reference crossing page boundary */
    if ((((i+j+1)%p1 - (i+j+2)%p1) >= 0) ||
        ((j%p2 - (j+2)%p2) >= 0)) {
      a = A[i+j+1][2*j] + B[i][j]
      flag = 1;
    } else {
      /* store buffered value */
      if (flag == 1) {
        A[i+j][2*j] = a;
        A[i+j+1][2*j+2] = A[i+j+1][2*j] + B[i][j];
        flag = 0;
      } else {
        /* normal computation */
        A[i+j+1][2*j+2] = A[i+j+1][2*j] + B[i][j];
      }
    }
  }
}
```

Only one page frame can be assigned to each array if they satisfy the above conditions. When there are references of multiple arrays, the same technique can be used for each individual array except that more flags are used.

5. Conclusion

In this paper, we have presented a set of compiler techniques which use either data layout or code transformations to improve paging performance for compiled programs. Algorithms for, and examples of, many of these techniques were given. Although we have not yet constructed a compiler implementing these transformations, the examples clearly demonstrate the potential benefits — as well as demonstrating how users can manually improve performance using existing compilers.

Ongoing work will continue toward formulation of a complete, consistent, view of compiler data layout and code transformations for improving paging performance. This should soon lead to a prototype compiler implementation of at least some optimizations, as well as simulation studies of performance. We have also begun investigating data layout issues for parallel systems

with physically distributed memory; many of the same concepts apply.

References

[Abu78] Abu-Sufah, W., *Improving the Performance of Virtual Memory Computers*, Ph.D. Thesis, University of Illinois at Urbana-Champaign, Nov. 1978.

[Abu79] Abu-Sufah, W., Kuck, D., Lawrie, D., "Automatic Program Transformations for Virtual Memory Computers," Proc. of AFIPS Nat'l Computer Conf. 1979, pp. 969-974.

[Abu81] Abu-Sufah, W., Kuck, D., Lawrie, D., "On the Performance Enhancement of Paging System through Program Analysis and Transformations," IEEE Trans. Computer, Vol. C-30, No. 5, May 1981, pp. 341-356.

[ACK87] Allen, R., Callahan, D., Kennedy, K., "Automatic Decomposition of Scientific Programs," Conf. Record of 14th Annual ACM Symposium on Principles of Programming Languages, 1987.

[AhU77] Aho, A.V., Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, 1977.

[BDM81] Budzinski, R., Davidson, E., Mayeda, W., Stone, H., "DMIN: An Algorithm for Computing the Optimal Dynamic Allocation in a Virtual Memory Computer," IEEE Trans. on Software Engineering, Vol. SE-7, No.1, Jan. 1981, pp. 113-121.

[CAC81] Chaitin, G.J., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., Markstein, P., "Register Allocation Via Coloring," Computer Languages, Vol. 6, 1981, pp. 47-57.

[ChD89] Chang, L-Y., and Dietz, H. G., ''An Introduction to Compiler Optimization of Data Layout,'' *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[ChH84] Chow, F., Hennessey, J., "Register Allocation by Priority-Based Coloring," Conf. Record of ACM SIGPLAN Symp. on Compiler Construction, 1984, pp. 222-232.

[Chi89] Chi, C-H., *Compiler-Driven Cache Management Using a State Level Transition Model*, Ph.D. Thesis, School of EE, Purdue University, May 1989.

[CLZ86] Cytron, R., Lowry, A., Zadeck, K., "Code Motion of Control Structures in High-Level Languages," Conf. Record of the ACM Symp. on Principles of Compiler Construction, 1986, pp. 70-85.

[CoD73] Coffman, E.G., Denning, P.J., *Operating System Theory*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[CyF87] Cytron, Ron, Ferrante, J., "What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation," International Conference on Parallel Processing, 1987, pp. 19-27.

[DeG75] Denning, P.J., Graham, G.S., "Multiprogrammed Memory Management," Proc. of the IEEE, Vol. 63, June 1975, pp. 924-939.

[Den68] Denning, P.J., "Thrashing: Its Causes and Prevention," Proc. AFIPS 1968 FJCC Vol. 33, pp. 915-922.

[Den80] Denning, P.J., "Working Sets Past and Present," IEEE Trans. on Software engineering, Vol. SE-6, No. 1, Jan. 1980, pp. 64-84.

[Die86] Dietz, Henry G., Principles of RISC Design for MIMDs, Tech. Report, CS Division, CS/EE Dept., Polytechnic Univ., 1986.

[Die87] Dietz, Henry, G., The Refined Language Approach to Compiling for Parallel Supercomputers, Internal Report, School of EE, Purdue University, May 1987.

[Els74] Elshoff, J.L., "Some Programming Techniques for Processing Multidimensional Matrices in a Paging Environment," Proc. of AFIPS Nat'l Computer Conf. 1974, pp. 185-193.

[Fab79] Fabri, J., "Automatic Storage Optimization," Proc. ACM SIGPLAN Symp. on Compiler Construction, 1979, pp. 83-91.

[Fer74] Ferrari, D., "Improving Locality by Critical Working Sets," Communication of ACM, Vol. 17, No. 11, Nov. 1974, pp. 614-620.

[Fer76] Ferrari, D., "The Improvement of Program Behavior," IEEE Computer, Nov. 1976, pp. 39-47.

[Fre74] Freiburghouse, R.A., "Register Allocation via Usage Counts," Comm. ACM Vol. 17, No. 11,Nov. 1974, pp. 638-642.

[Gue72] Guertin, R.L., "Programming in a Paging Environment," Datamation, Feb. 1972, pp. 48-55.

[HaG71] Hatfield, D.J., Gerald, J., "Program Restructuring for Virtual Memory," IBM System Journal, Vol. 10, No. 3, 1971, pp. 169-192.

[HwB84] Hwang, K., Briggs, F.A., *Computer Architecture and Parallel Processing*, Mcgraw Hill, 1984.

[LaF83] Lau, E.J., Ferrari, D., "Program Restructuring in a Multilevel Virtual Memory," IEEE Trans. on Software Engineering, Vol. SE-9, No. 1, Jan. 1983, pp. 69-79.

[KKL81] Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., Wolfe, M., "Dependence Graphs and Compiler Optimization," Conference Record of Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, VA., Jan., 1981.

[Kuh80] Kuhn, R.H., *Optimization and Interconnection Complexity for Parallel Processors, Single Stage Networks, and Decision Trees*, Ph.D. Thesis, Dept. of Computer Science, Report 80-1009, University of Illinois, Urbana-Champaign, 1980.

[Leu81] Leuze, Michael, *Memory Access Patterns in Vector Computers with Applications to Problems in Linear Algebra*,Ph.D Thesis, Duke University, Dec. 1981.

[Mac85] Mace, M., "Global Optimum Selection of Memory Storage Patterns," International Conference of Parallel Processing, 1985, pp. 264-271.

[Mac88] Mace, M., *Memory Storage Patterns in Parallel Processing*,Kluwer Academic Publishers, Norwell MA., 1988.

[MaP85] Malkawi, M., Patel, J., "Compiler-Directed Memory Management Policy for Numerical Programs," Proceeding of the Tenth ACM Symposium on Operating System Principles, Dec., 1985, pp. 97-106.

[MaP86] Malkawi, M., Patel, J., "Performance Measurement of Paging Behavior in Multiprogramming Systems," ACM SIGARCH Computer Architecture News, Vol. 14, No. 2, June 1986, pp. 111-118.

[Mar75] Martin, James, *Computer Data-Base Organization*, Prentice Hall, Englewood Cliffs, N.J., 1975.

[McK69] McKellar, A.C., Coffman, E., "The Organization of Matrices and Matrix Operations in the Paged Multiprogramming Environment," Comm. ACM., Vol. 12, No. 3, March 1969, pp. 153-165.

[Mol72] Moler, C.B., "Matrix Computation with FORTRAN and Paging," CACM vol. 15, No. 4, April 1972, pp. 268-270.

[MSN74] Masuda, T., Shiota, H., Noguchi, K., Ohki, T., "Optimization of Program Organization by Cluster Analysis," Information Processing 74 (Proc. IFIP Congress 74), North-Holland, Amsterdam, 1974, pp. 261-265.

[Smi78a] Smith, A.J., "Sequentiality and Prefetching in Data Base Systems," ACM Transactions on Data Base Systems, Vol. 3, No. 3, 1978, pp. 223-247.

[Smi78b] Smith, A.J., "Sequential Program Prefetching in Memory Hierarchies," IEEE Computer, Vol. 11, No. 12, 1978, pp. 7-21.

[Smi82] Smith, A.J., "Cache Memories," Computing Surveys, Vol. 14, No. 3, Sept. 1982, pp. 473-530.

[Triv77] Trivedi, K.S., "On the Paging Performance of Array Algorithms," IEEE Trans. Computer, Vol. C-26, No. 10, Oct. 1977, pp. 938-947.

[Wol82] Wolfe, M.J., *Optimizing Supercompilers for Supercomputers*, Ph.D Thesis, Univ. of Illinois at Urbana Champaign, 1982.