Compiler Directed Memory Management Policy For Numerical Programs

Mohammad Malkawi and Janak Patel

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, Il. 61801

Abstract

A Compiler Directed Memory Management Policy for numerical programs is described in this paper. The high level source codes of numerical programs contain useful information which can be used by a compiler to determine the memory requirements of a program. Using this information, the compiler can insert some directives into the operating system for effective management of the memory hierarchy. During the program's execution the operating system dynamically allocates to a program the space it requires as specified by the received directive. The new policy is compared with LRU and WS policies. Empirical results presented in this paper show that the CD policy can out-perform LRU and WS by a wide margin.

1 INTRODUCTION

Many memory management policies have been designed since the early invention of Virtual Memory (VM) systems. These policies assume either fixed memory allocation (static policies) such as Least Recently Used (LRU) and First In First Out (FIFO), or they assume variable memory allocation (dynamic policies) such as Working Set policy (WS) [Denn68] and Page Fault Frequency (PFF) [ChOp72].

Dynamic policies have been shown to outperform static ones [DeGr75], [BDMS81]. However, they have their own problems. The WS for example, is too expensive to implement; furthermore, it is unable to avoid heavy faulting rate during interlocality transitions [FeYi83]. The Damped WS (DWS) was introduced to handle these transitional faults [Smit76]. However, the DWS out performs WS by less than 10% [Grah76]. The Sampled WS (SWS) [RoDu73] is a cheaper realization of the WS . In [FeYi83] the Variable Sampled WS (VSWS) was proposed to reduce both implementation cost and transitional page faults. The Page Fault Frequency (PFF) is cheaper to implement [ChOp76] but has poorer performance than the WS [Grah76]; also, it exhibits anomalous behavior [FrGG78]. The WS also exhibits some types of anomalies when tested against numerical programs [AbPa81], [ALMY81]. Other types of WS anomalies in multiprogramming systems have been discovered by the authors of this paper. The controllability of the WS in a multiprogramming environment (10% de-tuned policy [GrDe78], [Denn80]) is too optimistic [ALMY82], [AbLM84]. The conclusions drawn about the WS optimality and controllability which were based mostly on experiments with systems programs do not hold for numerical programs [ALMY82], [AbLM84].

The difficulty of all of the above approaches is that they try their best to estimate the behavior of a program at run time. A fair amount of run time behavior can be predicted from the high level source code. However, none of the proposed memory management policies exploit that fact. Empirical results about the localities of numerical programs show that their localities can always be associated with iterative structures [Malk82]. It is not difficult to identify the localities of a program if the space required by its data structures is explicit in the source code, as is the case in most high level language programs. The information inherent in the source code is, therefore, helpful for specifying the memory requirement of a program at execution time.

In this paper we propose a memory management policy based on the recognition of locality characteristics at compile time. A study of the locality characteristics of FORTRAN numerical programs is presented in the next section. In Section 3 we develop the concept of memory directives (MD). An outline of a compiler-directed memory management policy, supported by MDs inserted into the program high level code, is presented in Section 4. Empirical results comparing CD with both WS and LRU policies are presented in section 5. Conclusions are drawn in Section 6.

2. LOCALITY CHARACTERISTICS OF NUMERICAL PROGRAMS

It has been observed very early that a program does not reference its address space randomly.

Acknowledgment: This research was supported by the Joint Services Electronics Program under Contract N00014-84-C-0149

Rather, it tends to reference a small subset of its space for a relatively long period of time. This property is referred to as the locality of reference [Denn72], [DeSp72],. Quantitatively, the localities can be characterized by three parameters. The length of a locality specifies the time duration, during which a locality exists. The virtual size of a locality specifies the number of distinct pages (in a paged system) comprising the locality set. Finally, the level of a locality specifies the depth of a locality in the hierarchical locality structure. The latter parameter is essential because several localities with different sizes and lengths can exist at the same time, thus, forming a hierarchical locality structure [MaBa76]. Studies of the locality characteristics of numerical programs have shown that the localities of these programs can be always associated with array references inside their loop structures [MaBa76], [Abus81], [Malk82]. Complicated loop structures yield complicated locality characteristics. In particular, multi nested loops produce hierarchical locality structure [Abus81], [Malk82]. Such hierarchical locality structure imposes some difficulties on the implementation of a memory mangement system which is designed to be aware of the existence of program localities. The example in Figure 1 illustrates the localities depicted in a FORTRAN like piece of code.

```
DO 10 I = 1, 10
  DO 20 J = 1, 200
     E(I,J) = F (I,J)
  20   CONTINUE

  DO 30 K = 1, 200
     G(K,I) = H(K,I)
  30   CONTINUE
10   CONTINUE
```

The localities of the above code are represented by the following diagram.

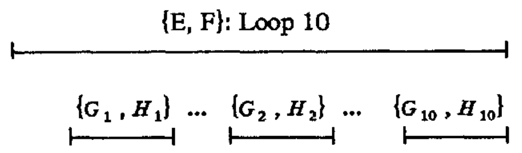

Figure 1: Example Of Localities At The Source Level

In this example, all four arrays (E,F,G,H) are repeatedly referenced in the outermost loop (loop 10). In loop 20, arrays E and F are referenced row wise while, in loop 30, arrays G and H are referenced column wise; the arrays are stored in a column major order scheme. Loop 20 does not form a locality, since none of the pages in the virtual spaces of arrays E and F are referenced repeatedly during the execution of loop 20. However, the same virtual space spanned by loop 20, is repeatedly spanned by loop 10. Therefore, arrays E and F form a locality at the higher level of loop 10; the size of this locality is the sum of the virtual sizes of arrays E and F. During the execution of loop 30, one column from array G and one column from array H are referenced repeatedly, therefore, forming a locality at the lower level of loop 30.

The above example is too simple to illustrate the complexity involved in identifying all possible localities of a program at a given time. It is not, however, the intention of this paper to introduce a systematic procedure for computing the virtual size of program localities at the source level code. Rather, we present a study of the various parameters which could be used for this purpose. We examined a wide range of FORTRAN programs used in different packages for the purpose of identifying and measuring their localities using the information inherent in the source code. Some of these packages are UIARL : University of Illinois Atmospheric Research Lab, EISPACK, ACM: ACM Standard Programs, IEEE: IEEE Standard Programs; NRL: Naval Research Laboratory, AFWL: Air Force Weapons Laboratory, FISHPACK, MINPACK.

Examining the source code of these programs reveal that six parameters can be used to calculate the virtual size of current program localities. Five of these parameters are program dependent and one is system dependent. The system dependent parameter is the page size (P). The page size is necessary for calculating the virtual size of the current locality in pages since memory allocation is measured in pages. Program dependent parameters are:

(1) Array size (Σ) : Σ is usually given as (M x N) dimension, where M = number of rows and N = number of columns. N = 1 for vectors. Only up to two dimensional arrays are considered in this paper. Array sizes are given explicitly in the dimension declaration statements. The virtual size of an array (AVS) is given by:

$$AVS = \frac{(M \times N)}{P}$$

Often the virtual size of an array column (CVS) contributes to the size of the current locality since; CVS is given by:

$$CVS = \frac{M}{P}$$

The virtual size of all arrays referenced in a program comprise an upper bound on the memory requirement of the program. Memory requirements during the execution interval of a loop structure are bounded by the virtual size of the arrays referenced inside the structure. We assume that all constants and instructions are permanently resident in memory. Finer calculation of locality sizes is facilitated by other parameters described below.

(2) The nest depth of the loop structure (Δ): Δ determines whether the current locality has a hierarchical structure or not. $\Delta > 1$ implies that a hierarchical locality structure with utmost Δ levels may exist. It is possible, however, not to have a hierarchical locality structure with $\Delta > 1$. A typical example is a doubly nested loop, $\Delta = 2$, with arrays referenced row-wise inside the inner loop. In this case, the doubly nested loop forms a single level one locality. Δ is also used for assigning *priority indexes* (defined in the next section) to the current loop.

(3) The number of indexed variables used to reference array elements (X): X is used to give an upper bound on the number of distinct array pages referenced at a given locality level. The maximum number of array elements which can be referenced during one iteration of a loop is determined by the number of distinct indexes, X, used to address the array. If the array elements referenced at a particular level are stored in distinct pages, then X distinct pages are referenced at this level. Depending on the dimension and the order of reference of an array, X can be used to give a real upper bound on the number of array pages which participate in the formation of the locality at the current level. For the case of a vector the upper bound is found by counting the distinct indexed variables used to index the vector. For example, consider the following statement involving a vector V inside a loop:

$$W = V(I) + V(I+1) + V(J);$$

Three different indexes are used to reference vector V; namely I, I+1 and J. Obviously, a maximum of three pages of vector V can be referenced during one iteration of the loop containing V. For column-wise referenced arrays , where the column size is larger than a page size, an upper bound is given by

$$X = X_c \times X_r$$

where X_r is the number of distinct indexes used to reference a column ($r \leqslant$ the number of rows M) and X_c is the number of distinct columns referenced at the current locality level ($c \leqslant$ the number of array columns N). Consider for example, the following statement encountered in an inner loop:

$$W = A(I,J) + A(I+1,J) + A(I,J+1) + A(I+1,J+1)$$

where I is the index of the inner loop and J is the index of the outer one. In this example, J and J+1 are two different columns referenced inside the inner loop. I and I+1 are used to reference elements in columns J and J+1. Four elements of array A are referenced during one iteration of the inner loop ; the four elements can be stored in four pages at most. If the array is referenced row-wise, then the maximum number of pages referenced during one iteration of the loop containing the array is given by:

$$X = X_r \times N$$

where N is the number of columns in the array. We use N here, instead of X_c, because X_c is given by the loop's upper bound, a variable which might not be known at compile time. Therefore, we assume that once a row I is referenced, all of its elements (I,1), (I,2), ... ,(I,N) will be referenced as well. However, if the total number of array pages AVS is larger than X, then the upper bound is determined by AVS.

(4) The order of reference (Θ): The order in which arrays are referenced has a direct effect on the formation of a locality. If an array is referenced column-wise, then the referenced columns participate in the formation of the locality comprised by the loop containing the array. On the contrary, a row-wise referenced array tends to have most of its virtual space spanned during the whole duration of the loop in which it is referenced. Therefore, the pages referenced during one iteration of the loop are not likely to be referenced during the next iteration. Row-wise referenced arrays tend to form localities at higher levels than the ones they are referenced at.

(5) The level (or nest depth) at which the arrays are referenced (Λ): $\Lambda = 1$ for the outermost loop in a multi nested loop structure. Λ increases for loops as we go deeper into the nest. $\Lambda = \Delta$ for the innermost loop. The smaller the value of Λ, the higher the level. A row-wise referenced array at some level $\Lambda = \lambda$ tends to form a locality at higher levels $\Lambda > \lambda$ because the same virtual space of the array might be spanned at level λ during one iteration of a loop executing at level $> \lambda$. Similarly, for the case of a vector, one iteration of a higher level loop is sufficient to span the entire virtual space of all vectors referenced at lower levels. Therefore, the entire virtual space of a vector referenced at level $\lambda \neq 1$ contributes to all higher level localities. In the case of a column-wise referenced array inside a loop at level $\Lambda = \lambda$, one or more columns of the array are spanned during the execution of the loop. These columns are usually specified by an outer loop at level $\lambda - 1$. The entire virtual space is spanned during one iteration of a loop at level $\lambda - 2$. Thus the entire virtual space of a column-wise referenced array contributes to localities formed at least two levels higher than the level at which the array is referenced.

For this study, we use the above parameters in a non-deterministic manner to evaluate program localities. A deterministic procedure is being developed by the authors of this paper.

3. MEMORY DIRECTIVES

The above discussion leads to the idea of memory directives (MD) as a tool for improving the memory management system. A memory directive is an instruction whose execution generates useful information used by the operating system for memory allocation and deallocation purposes. MD's are to be inserted into the program code before execution time. Madison and Batson introduced this idea and developed the BLI model to characterize program localities [MaBa76] with the intention of making these localities known to the operating system at execution time. In [Abus81] and [Malk82] one further step was taken in this direction. The BLIs were shown to reflect programs' iterative structures. Two directives were suggested to allocate and deallocate memory space on behalf of users' programs. The idea of using memory directives was also advocated by Hagman and Fabry [HaFa83] and Kearns and Defazio [KeDe83]. Some operating systems even provide facilities to improve program behavior in virtual memory systems using the idea of memory directives. For instance, VAX/VMS allows the user to lock and unlock certain pages in the main memory. Such facilities were shown to be useful for improving the behavior of some numerical algorithms [Abaz84].

The characteristics of MD, where and how to insert them into the program code, and how to process a MD by the system will be presented in the next two subsections.

Three memory directives are investigated in this section: *LOCK*, *UNLOCK* and *ALLOCATE*. Both directives LOCK and UNLOCK have been used in recent operating systems as mentioned above. In these systems it is left to the user to decide which pages to lock or unlock at some given time. In this paper we present a procedure to resolve the problem of automatic insertion of LOCK and UNLOCK directive. The effectiveness of LOCK and UNLOCK directives is not studied in this work.

The ALLOCATE directive is used to allocate physical memory on behalf of the running process. ALLOCATE does not deal with particular pages as LOCK and UNLOCK do. To the best of our knowledge, the effectiveness of such a directive has not been investigated. Some researchers predicted the viability of such a directive due to program locality characteristics [MaBa76], [AbLM84]. In this study we develop the concept of MD for a multiprogramming environment, discuss implementation issues of the directives, and present some empirical results on program behavior under a memory management policy guided by the ALLOCATE directive.

3.1. Memory Directive : ALLOCATE

The memory allocation directive is used for estimating the memory space required by a program during its execution. ALLOCATE is a measure of the virtual size of the current locality. In developing ALLOCATE for a multiprogramming system, two facts were considered. First, the available memory in a multiprogramming system dynamically changes as processes acquire and release memory pages. Second, memory requirements of a program change dynamically according to the program's current locality characteristics. The locality characteristics include the size of the locality and its level in the locality hierarchy. Program localities exhibit hierarchical structure [MaBa76] which reflects the nested loop structures exhibited at the source level code [Abus81], [Malk82]. The decision as to which level of the hierarchy the directive should be applied is of special significance. In [BaBK77] a structure parameter α was introduced to decide whether the lower level BLIs are significant enough to consider for memory allocation. However, α is a system dependent parameter [BaBK77]; furthermore, it can not respond to the dynamic change of memory space availability due to multiprogramming. Calculating the structure parameter α further complicates the memory directive's implementation.

To handle the effect of changes both in a program's memory requirements and in the available memory due to multiprogramming, we propose a *priority index* (PI) to be used in the argument list of ALLOCATE. Each ALLOCATE directive is capable of issuing several memory requests. Each request is assigned a priority index. The memory directive ALLOCATE has the following form:

ALLOCATE $((PI_1\ ,\ X_1)$ else $(PI_2\ ,\ X_2)$ else ...)

where

X_i is the number of memory pages requested by the program. X_i corresponds to the virtual size of the current locality.

$PI_1 > PI_2 > PI_3 > \ldots$

$X_1 \geqslant X_2 \geqslant X_3\ \ldots$

The priority index tells the system whether the program is approaching a single locality or a hierarchy of localities, in which case the maximum level in the hierarchy is determined by the maximum PI. Furthermore, PI imposes a priority on the order of receiving the requests issued by the ALLOCATE directive. Larger memory requests have larger values of PI and they are tried for allocation first. Moreover, the system can tell, using PI, which ALLOCATE requests may not be granted at the moment of their execution and which requests have to be satisfied. For this purpose the smaller the value of PI the higher the priority. One can come up with a complicated priority structure from the analysis of the source program, the available memory and the number of processes competing for memory. In this paper we have chosen a priority structure with the following properties:

(1) The highest priority, PI = 1 is associated with

the inner most loops.

(2) The lowest priority, PI = Δ is associated with the outer most loop.

(3) The priority associated with any level in between the inner most and the outer most loops depends on its level, Λ, in the loop nest and its distance from the inner most loop for which the loop at level Λ is an outer loop.

The priorities are assigned to the various levels of the loop structure according to the procedure given in Figure 2. This procedure scans all the loops in a bottom-up manner. In this scheme the priority index associated with level Λ, is given by the distance from Λ to the inner most level of the loop substructure involving Λ. An example is given in Figure 2.

```
Procedure 1: Assign Priority Indexes;
  With every inner loop in the nested loop structure
          DO
  Assign PI = 1 to the inner most loop;
  REPEAT
  Next Outer Loop;
  IF (PI is already assigned)
     THEN   PI = maximum (PI+1 , old PI);
     ELSE   PI = PI + 1;
  UNTIL Outer Most Loop Is Encountered;
          END Of Procedure;
```

Example:

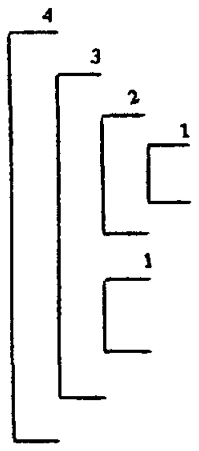

Figure 2: Procedure and Example of Assigning Priority Indexes

The virtual size X, of a locality formed by a loop at some level in a multi nested loop, can be computed using the parameters given in section two.

The arguments of ALLOCATE at some level Λ = λ, are carried out at all subsequent levels > λ. This technique allows requests not granted because of lack in memory space to be reconsidered at later periods. It also favors the outer loops with larger locality sizes which aim at reducing the number of page faults generated by outer loops.

The ALLOCATE directive is inserted into the program code according to algorithm 1 given in Figure 3.

```
Algorithm 1:

 INITIALIZE:   MD = ALLOCATE ();
          {List is empty}
 PARSE
          {until the end of the program}
 CASE of  encountering a loop  DO :
    Current PI = PI associated with current loop;
          {Procedure 1 in Figure 2}
    Current X  =  The virtual size of the current
          locality;
    IF the argument list of MD is empty
       THEN   APPEND (Current PI, Current X) to
          MD list
       ELSE   APPEND else (Current PI, Current X)
          to MD list;
    INSERT MD    {right before the beginning of the
          loop};
 END of CASE statement;
 CASE of exiting a loop  DO:
    DELETE last two elements of the argument list
          of MD;
END of Algorithm 1.
```

Figure 3: Algorithm For Inserting the ALLOCATE Directive

In algorithm 1 a single top-down parsing procedure is used to insert the memory directive ALLOCATE into the program's code. ALLOCATE is inserted before the beginning of every loop in the program. We use a list structure to represent the memory directive, MD. The head of the list is the element ALLOCATE (directive name); the rest is the argument list consists of elements of the form (PI,X) separated by the element "else". When the parser discovers a loop, a new MD is generated. An MD at any level is updated by appending the element pair "else (PI,X)", at the tail of the MD argument list. PI is the priority index associated with the current loop and X is the virtual size of the locality comprised by the current loop. In case the current loop does not form a locality, X is evaluated to the minimum number of pages which a program is allocated by system default.

Upon exiting a loop, the MD argument list is updated by deleting the last "else (PI,X)" elements from the list, since any loop discovered in the future can not be enclosed by the exiting one. Therefore the arguments of the exiting loop will not be present in the argument list of the next generated directives. By deleting the arguments of the exiting loop we avoid backtracking when generating an MD for the next loop in the program.

3.2. LOCK and UNLOCK Memory Directives

LOCK is used to prevent some pages from being paged out of memory by the replacement policy. UNLOCK relases these pages. Pages to be locked are usually vectors or array columns referenced inside outer loops. References to pages of these arrays are expected to generate faults whenever the execution of an inner loop is finished and a branch is made back to an outer loop. LOCK is useful when the allocation request made by an ALLOCATE directive associated with an outer loop, is not granted. In such cases, locking those pages referenced in the outer loop which might be rereferenced avoids a possible increase in the number of page faults. In case of high memory contention the operating system is entitled to release the locked pages without having to wait for the UNLOCK directive to be executed. This flexibility makes the LOCK directive a soft one. The order of releasing pages by the system, without using UNLOCK, is determined by a priority index PJ similar to PI for ALLOCATE. PJ is calculated using procedure 1, given in Figure 2. Since there will be no pages locked in the inner most loop, where the priority index is 1, the highest priority of locked pages is PJ = 2. Pages locked inside the outer most loops generate less faults than those referenced inside inner loops since they have lower frequency of reference. Therefore, pages with higher PJ values have lower priority and they are unlocked first by the operating system. The LOCK directive has the following form:

LOCK (PJ, Y_1, Y_2, ...)

where

PJ is the priority index and
Y_i is the particular page to be locked in memory. In case there are no pages to be locked, $Y_i = 0$.

The LOCK directive is inserted into the program code according to algorithm 2, given in Figure 4.

```
Algorithm 2  {for a given loop structure}

  PARSE
     {until the end of the outer most loop}
  CASE of encountering a loop DO
  PJ = The priority associated with the current loop;
  SEARCH for arrays until the next loop is
            discovered;
  IF Loop Exit Is Found Then SKIP Next INSERT;
  $Y_i$ = Array Page To Be Locked (i = 1,2,...);
   INSERT   LOCK (PJ, $Y_1$ , $Y_2$ , ...);
     {before the beginning of next loop}
```

Figure 4: Algorithm For Inserting "LOCK" Directive

To unlock the locked pages UNLOCK is inserted at the end of the outer most loop. The unlock directive has the following form:

UNLOCK (Y_1, Y_2 ...)

where Y_1, Y_2... are the pages which were locked by the LOCK directive.

```
DO 4 I=1,N
  A(I), B(I)

  DO 2 J =1,N
    C(J) , D(J),
    CC (I,J), DD (J,I)
  2 CONTINUE

  DO 3 K =1,N
    E(K) , F(K)

    DO 1 L =1,N
      X(L), Y(L), Z(L),
      XX (K,L), YY (L,K)
    1 CONTINUE
  3  CONTINUE
4   CONTINUE
```

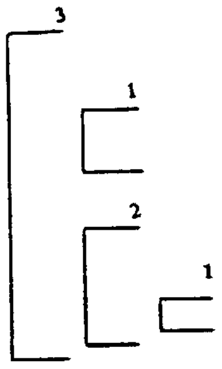

Figure 5a: FORTRAN-Like Code   Figure 5b

```
ALLOCATE (3,X1)
Loop 4;
|
|  LOCK (3,A,B)
|  ALLOCATE (3,X1) else (1,X2)
|  Loop 2;
|  |
|  |_
|  ALLOCATE (3,X1) else (2,X3)
|  Loop 3;
|  |
|  |  LOCK (2,E,F)
|  |  ALLOCATE (3,X1) else (2,X3) else (1,X4)
|  |  Loop 1;
|  |  |
|  |  |_
|  |_
|_
UNLOCK (A,B,E,F)
```

Figure 5c: Directives Inserted Into The Code of Figure 5a

The directives' insertion is illustrated through the example given in Figure 5. Figure 5a shows a piece of FORTRAN like code. In Figure 5b priority indexes are assigned to the loops in Figure 5a, according to the procedure in Figure 2. In Figure 5c the directives ALLOCATE, LOCK and UNLOCK are inserted into the code of Figure 5a. Note that the argument (3,X_1) is the first argument in all ALLOCATE directives at all levels. (2, X_3) is present in the directives associated with loop 3 and its inner loop (loop 1). A subscripted X denotes the virtual size of the locality comprised by the loop before which the directive is inserted. Consider for example, the directive ALLOCATE inserted before loop 4. PI = 3 since loop 4 is the outer most loop and $\Delta = 3$.

The virtual size of the locality formed by loop 4, X_1, is computed by considering all the arrays encapsulated by loop 4. Consider the arrays referenced only inside loop 4, at level $\Lambda = 1$. There are two vectors, A and B, referenced at this level. One indexed variable is used for each vector. Allocating one page for each vector will be sufficient during the execution of loop 4, since once a new page of A or B is referenced, the old one will not be referenced again. At level two four vectors and two arrays are referenced. Vectors C and D are referenced inside loop 2 and vectors E and F are referenced inside loop 3. The entire virtual size of every one of these vectors will be spanned N times; N is the upper bound of loop 4. Hence, the entire virtual sizes of C, D, E and F contribute to the locality size at level 1. As for the arrays CC and DD we examine their referencing order. Array CC is referenced row-wise. Since the arrays are stores in a column major order, loop 2 spans all the pages in which the elements of row I are stored. One indexed variable I is used to address the array CC; therefore, $X_r = 1$. The maximum number of CC pages which could be referenced inside loop 2 is given by N * 1 = N, (CC is N x N array). Thus CC contributes to the value of X_1 with N pages. Array DD is referenced column-wise. Every time loop 4 iterates, a new column I is referenced by loop 2. Since there is only one indexed variable, J, used for indexing the column I, there is only one active page of array DD during the execution of both loops 2 and 4. Array DD thus contributes to X_1 with one page only. At level 3, all of the arrays, regardless of their dimension or their order of reference, participate in the formation of the level one locality with their entire virtual sizes.

LOCK (3, A, B) is used to lock a page of A and a page of B. This LOCK is useful if the request of ALLOCATE (3,X_1) is not granted. Locking pages of arrays E and F is useful if the request made by ALLOCATE (3,X_1) else (2,X_3) at the beginning of loop 3 is not granted. The UNLOCK directive is used to unlock all the pages which were locked by the LOCK directive in case they we were not released by the operating system.

4. COMPILER DIRECTED MEMORY MANAGEMENT POLICY

The memory directives described by algorithms 1 and 2 in the previous section can be incorporated into an optimizing compiler to generate MD for the operating system (OS). The OS uses these directives for memory management purposes. The resulting *Compiler Directed Memory Management Policy* (CD) works as follows. At execution time the CPU interprets these directives as calls to the operating system. If the call is generated by a LOCK directive, the operating system locks the pages specified in the argument of the LOCK directive. These pages are kept locked until a UNLOCK directive is processed and a call is generated to the OS. However, the OS is entitled to release locked pages without receiving UNLOCK, in case of high memory demands. The priority index PJ is used to decide which pages should be released first.

Upon receiving a call generated by a directive of the form *ALLOCATE* ((PI_1 , X_1) else (PI_2 , X_2) else...), the operating system allocates X_1 pages to the program if X_1 pages are available, otherwise it allocates X_2 if available, ($X_2 < X_1$ and $PI_1 > PI_2$), etc. If the requested pages by the ALLOCATE directive are not available and the smallest priority index specified by the directive $PI_j = 1$, then the operating system either suspends the program's execution or invokes a page swapping procedure in case the current job has a higher priority. PI > 1 means that the current locality is comprised by one of the outer loops in a multi-nested loop. The operating system in this case continues the execution of the program with the current allocation until it receives a new directive. This procedure continues until a directive with PI=1 is reached. A flowchart describing this procedure is given in Figure 6.

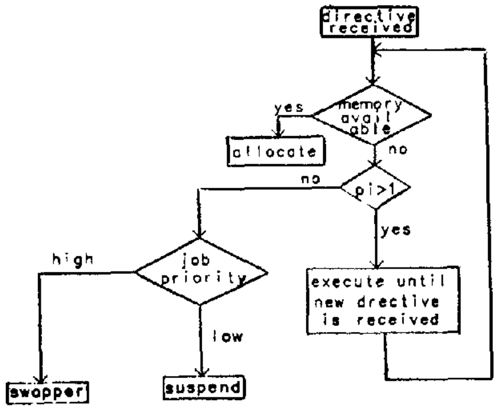

Figure 6: Memory Directives Processing

CD policy incorporates a swapping mechanism. None of the existing memory management policies has this feature. The WS policy [Denn68] incorporates a mechanism only for invoking the swapper. However, the WS does not provide the swapper with any useful information regarding the processes to be swapped. CD policy invokes the swapping mechanism whenever a memory request associated with a priority index PI = 1 can not be granted. The swapper is never invoked by a request whose priority is > 1. This implementation reflects the fact that with PI > 1, the program can reside in several localities of different sizes at various levels of the hierarchical locality structure.

5. EXPERIMENTS AND RESULTS

In our experiments we assume a paged system with a 256 byte page size. The performance indexes used in this paper are the number of page faults (PF), the average memory allocated to a program (MEM), and the space time cost (ST). ST includes the time for page fault service, assumed to be 2000 memory references.

Traces of array references were generated for 9 numerical programs written in FORTRAN. A virtual memory simulator is used to simulate program behavior under the Least Recently Used (LRU), the Working Set (WS), and the CD policies. The simulation is done for a uniprogramming system. The performance of CD in a multiprogramming environment is still to be evaluated.

In this paper we compare CD with both LRU and WS. The WS parameter, the window size τ, is varied between 1 and some integer $K \leqslant R$, the reference string length. For LRU the memory allocated to a program is varied between 1 and V, where V is the virtual size of the program measured in pages. For CD policy the memory directives inserted at the source level before program execution determine the average memory allocated to the program, the number of page faults generated, and the space-time cost. In our experiments we specify prior to program execution the set of directives to be executed, since we assume no physical limit on the available memory. To produce different values, a program has to be rerun with different sets of MD. Programs MAIN, FDJAC and TQL were rerun with different sets of directives. In a multiprogramming environment the priority index PI dynamically determines which set of directives to execute.

The excess memory that LRU or WS require over CD to achieve a given performance goal is given by:

$$\%MEM = \frac{MEM\,(LRUorWS) - MEM\,(CD)}{MEM\,(CD)} 100$$

Similarly, excess space-time cost that LRU or WS produce over CD is given by:

$$\%ST = \frac{ST\,(LRUorWS) - ST\,(CD)}{ST\,(CD)} 100$$

and excess page faults that LRU or WS produce over CD is given by:

$$\Delta PF = PF\,(LRUorWS) - PF\,(CD)$$

In Table 1 we show the effect of executing different sets of directives on the performance of the CD policy. Four sets were used for program MAIN and two sets for each of FDJAC and TQL. Less memory allocation results from executing the directives associated with the inner loops. Directives at outer levels consume more memory and generate fewer page faults. This reflects the fact that the sizes of lower level localities is always smaller than those at the higher levels. For example, when program MAIN1 was executed with the directives at the outer most levels, 144 page faults were generated and 3.89 x 10^6 space-time cost was produced. When the program was executed with the directives inserted at the lower levels (MAIN3), the number of page faults increased 4.5 times (652 page faults) while the ST cost dropped by 50% (2.77 x 10^6); this is lower than the minimum ST cost under the WS by 17% and under LRU by 47%. Similarly for programs FDJAC and TQL the lower level directives produced less ST cost than did the WS by 39% and less ST cost than did the LRU by 28%. Our observation holds for the rest of the programs. See Table 2.

Table 1: The Effect of Executing Different Sets of Directives Under CD Policy

Program	MEM	PF	$ST_{\min}$ (10^6)
MAIN	1.62	531	3.39
MAIN1	20.37	144	3.89
MAIN2	12.23	319	10.6
MAIN3	1.11	652	2.77
FDJAC	2.47	178	1.46
FDJAC1	3.11	175	2.04
TQL1	2.48	322	2.84
TQL2	2.02	421	3.063

Table 2: Comparing Minimal Space Time Cost Values of LRU and WS versus CD

	%ST	
PROGRAM	LRU vs. CD	WS vs. CD
MAIN3	47	17
FDJAC	27	39
FIELD	23	6
INIT	133	22
APPROX	36	58
HYBRJ	31	32
CONDUCT	288	32
TQL1	07	04

Next we compare the performance of the three policies by allocating the programs the same memory space. The corresponding number of page faults and the ST cost are compared. Since CD policy produces only one value for each performance index, while LRU and WS produce many values, as discussed earlier, we chose to select the average memory allocated by CD. Similar values were obtained by direct assignment for LRU or by adjusting the WS parameter, the window size τ. Comparisons are presented in Table 3.

For example, the program CONDUCT has a total of 270 pages in its virtual space. CD policy allocates on the average 25.8 pages and produces 577 page faults and 20.5 x 10^6 ST cost. Using 26 pages, LRU policy produces 3477 more page faults and has a larger ST cost by 988.3%. The WS uses a window size $\tau = 421$ to allocate on the average 25.7 pages to program CONDUCT. Using this τ the WS produces 1944 more page faults and has a larger ST cost by 1850.5%. From this table it is clear that CD policy makes best use of the memory space available for these programs. Using the same amount of memory, LRU and WS produce on the average 2863 and 2340 more page faults than does CD. ST cost reduction achieved by CD on the average is a factor of 5.24 and 5.57, compared with LRU and WS respectively.

Table 3: Comparing LRU and WS versus CD When Similar Average Memory is Allocated to All Policies

	LRU vs. CD		WS vs. CD	
PROGRAM	*ΔPF*	%ST	*ΔPF*	%ST
MAIN	1530	146.3	0	-4.7
MAIN1	236	338.87	207	316.45
MAIN2	207	35.5	207	19.8
MAIN3	22665	1585.9	22665	1585.9
FDJAC	337	115.75	293	91.1
FDJAC1	53	-6.8	296	60.78
FIELD	2643	1538.9	2	18
INIT	2287	979.5	775	630
APPROX	365	54.3	203	83.5
HYBRJ	317	159.1	283	139.1
CONDUCT	3477	988.3	1944	1840.5
TQL1	1017	191.55	958	223.9
TQL2	918	170.6	969	214.4
HWSCRT	4028	1047.9	4033	2265.2

One more way of comparing LRU and WS versus CD is to compare the three policies' memory and ST costs of producing the same number of page faults. Again we choose the number of page faults generated by CD for the same reason mentioned earlier. Comparisons are presented in Table 4.

Table 4: The Cost of Generating The Same Number of Page Faults as CD by LRU and WS

	LRU vs. CD		WS vs. CD	
PROGRAM	%MEM	%ST	%MEM	%ST
MAIN	150	32	14	-4.7
MAIN1	170	415.68	72.5	216.45
MAIN2	88	58	80.5	49.5
MAIN3	170.3	46.6	64	16.6
FDJAC	102	26.7	123	39
FDJAC1	60.7	-9.3	77	-0.3
FIELD	106.8	29.5	53.4	28
INIT	171.2	132.5	151.8	108.2
APPROX	105.8	36.2	34.4	77.9
HYBRJ	41.5	29.5	82.3	140
CONDUCT	283.7	324.6	11.6	36.1
TQL1	61.3	34.8	86.4	4.2
TQL2	98	25.2	128.8	-3.3
HWSCRT	442	433.5	124.6	234.3

In this table program HWSCRT has 69 pages in its virtual space. It generates 521 page faults using 11.8 pages on the average and a ST cost of 9.53 x 10^6 when CD policy is used. LRU needs at least 63 pages of memory, 442% more than CD needs, to generate at most 521 page faults. Excess ST cost (%ST) is 433.5%. The WS policy needs 124.6% more memory and 234.3% more ST cost than CD to generate at most 521 page faults. Table 7 shows that CD out performs both LRU and WS by a great margin. LRU and WS need on the average 247% and 175% respectively, more memory than the CD needs to generate the same number of page faults. The average excess ST cost (%ST) is 216.45% and 55%, for LRU and WS, respectively.

5. CONCLUSIONS

A new approach to the management of numerical programs in virtual memory systems is presented in this paper. Numerical programs exhibit localities which are governed by data references inside loop structures. The source level code of numerical programs contains sufficient information for locality identification purposes. Memory Directives can be inserted into the source code to identify and describe the localities of the program to the operating system. These directives are mainly used to specify the program's memory demand at execution time. Three memory directives were discussed in this paper, LOCK, UNLOCK and ALLOCATE. We presented algorithms for automatically inserting these directives into the program source code. A compiler directed memory management policy (CD) was also introduced in this paper. The main features of CD policy include: First, CD is able to dynamically adjust a program's memory allocation according to the status of the available memory on the system, which dynamically changes as processes acquire and release memory space. Second, it incorporates a swapping mechanism. Inserting directives into the programs source code to guide the operating system memory manager significantly improves the performance of the virtual memory system. Tables 2, 3 and 4 present evidence of the improvement of the compiler directed policy. CD, compared to WS and LRU, makes better use of the memory allocated to the program to produce many fewer page faults and to achieve lower space-time cost.

REFERENCES

[Abaz84] M. Abaza, "On The Effectiveness Of Memory Management System Calls In VAX/VMS," M.S. Thesis, Yarmouk University, Depart. of Electrical Eng., October 1984.

[Abus82] W. Abu-Sufah, "Identifying Program Localities at The Source Level," University of Illinois, Dept. of Comp. Science, Report No. UIUCDCS-R-82-1108, October, 1982.

[AbLM84] W. Abu-Sufah, R. Lee and M. Malkawi, "Identifying Two Program Categories for Memory Management Purposes," *Proc. of the 1984 IEEE 8th International COPMSAC*, pp. 492-503, November, 1984.

[AbPa81] W. Abu-Sufah and D. A. Padua, "Some Results on the Working Set Anomalies in Numerical Programs," *IEEE Trans. on Software Engineering*, Vol. SE-8, No. 2, pp. 97-106, March 1982.

[AhDU71] A. V. Aho, P. J. Denning and J. D. Ullman, "Principles of Optimal Page Replacement," *J. ACM 18* pp. 80-93, January 1971.

[ALMY81] W. Abu-Sufah, R. Lee, M. Malkawi, and P-C. Yew, "Empirical Results on the Behavior of Numerical Programs in Virtual Memory Systems," University of Illinois, Dept. of Computer Science, Report No. UIUCDCS-R-81-1076, November 1981.

[ALMY82] W. Abu-Sufah, R. Lee, M. Malkawi, and P-C. Yew, "Experimental Results on the Paging Behavior of Numerical Programs," *Proc. of the 6th International Conf. on Software Engineering*, pp. 110-117, September 1982.

[BaBK77] A. P. Batson, D. W. E. Blatt, and J. P. Kearns, "Structure Within Locality Intervals," in *Measuring, Modelling and Evaluating Computer Systems*, H. Beilner and E. Gelenbe, Eds., North-Holland Publishing Company, 1977.

[BDMS81] R. Budzinski, E. Davidson, W. Mayeda, and H. Stone, "DMIN: An Algorithm for Computing the Optimal Dynamic Allocation in a Virtual Memory Computer," *IEEE Trans. on Software Engineering*, Vol. SE-7, No. 1, pp. 113-121, Jan. 1981.

[ChOp72] W. W. Chu and H. Opderbeck, "The Page Fault Frequency Replacement Algorithm," in *1972 AFIPS Conf. Proc., Fall Joint Comput. Conf.*, Vol. 41, AFIPS Press, pp. 597-609, 1972.

[ChOp76] W. W. Chu and H. Opderbeck, "Program Behavior and the Page Fault Frequency Replacement Algorithm," *Computer*, Vol. 9, No. 11, pp. 29-38, Nov. 1976.

[CoRy72] E. G. Coffman and T. A. Ryan, "A Study of Storage Partitioning Using a Mathematical Model of Locality," *Commu. ACM 15, 3* pp. 185-190, March 1972.

[Denn68] P. J. Denning, "Working Set Model for Program Behavior," *Comm. of the ACM*, Vol. 11, No. 5, pp. 323-333, May 1968.

[Denn80] P. J. Denning, "Working Sets Past and Present," *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 1, pp. 64-84, Jan. 1980.

[DeGr75] P. J. Denning and G. S. Graham, "Multiprogrammed Memory Management," *Proc. of the IEEE*, Vol. 63, pp. 924-939, June 1975.

[DeKa75] P. J. Denning and K. C. Kahn "A Study of Program Locality and Life-time Functions," *Proc. 5th Symp. Operating Systems Principles, ACM SIGOPS*, pp. 207-216, Nov. 1975.

[Denn72] P. J. Denning, "On Modeling Program Behavior," *Proc. AFIPS SJCC*, pp. 937-945, 1972.

[DeSp72] P. J. Denning and J. R. Spirn, "Experiments with Program Localities," *AFIPS FJCC*, pp. 611-621, 1972.

[FeYi83] D. Ferrari and Y-Y. Yih, "VSWS: The Variable-Interval Sampled Working Set Policy," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 3, May 1983.

[FrGG78] M. A. Franklin, G. S. Graham, and R. K. Gupta, "Anomalies with Variable Partition Paging Algorithms," *Comm. of the ACM*, Vol. 21, No. 3, pp. 232-236, Mar. 1978.

[Grah76] G. S. Graham, "A Study of Program and Memory Policy Behavior," Ph.D. thesis, Purdue Univ., Dept. of Computer Science, Dec. 1976.

[GrDe78] G. S. Graham and P. J. Denning, "On the Relative Controllability of Memory Policies," in *Computer Performance*, K. M. Chandy and M. Reiser, Eds. Amsterdam, The Netherlands: North-Holland, pp. 411-428, Aug. 1977.

[HaFa82] R. B. Hagman and R. S. Fabry, "Program Page Reference Patterns," *Proc. of the 1982 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 20-29, August 1982.

[HaPo83] H. J. Haikala and H. Pohijanlahti, "On the BLI-Model of Program Behavior," *Proc. of the 1983 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 28-38, August 1983.

[KeDe83] J. Kearns and S. DeFazio, " Locality of Reference in Hierarchical Database Systems," *IEEE Trans. on Software Eng.*, Vol.SE-9, No. 2, March 1983.

[MaBa76] A. W. Madison and A. P. Batson, " Characteristics of Program Localities," *Comm. of the ACM*, Vol. 19, No. 5, pp. 285-294, May 1976.

[Malk82] Mohammad Malkawi, "Some Aspects Of Numerical Program Behavior In Virtual Memory Systems,"; M.S. Thesis, Department Of Electrical Engineering, Yarmouk University, Jordan.

[RoDu73] J. Rodriguez-Rosell and J. P. Dupuy, "The Design, Implementation and Evaluation of a Working Set Dispatcher," *Commun. of the ACM*, Vol. 16, pp. 556-560, Sept. 1973.

[ScTu72] G. S. Schedler and C. Tung, "Locality in Page Reference Strings," *SIAM J. on Computing 1, 3* pp. 218-241, Sept. 1972.

[Smit76] A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Trans. on Computers*, Vol. C-25, No. 9, pp. 907-914, September 1976.

[Spir76] J. R. Spirn, "Distance String Models for Program Behavior," *Computer 9, 11* pp. 14-20, Nov. 1976.