

Schedule-Independent Storage Mapping for Loops

Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Beth Simon
{mstrout,carter,ferrante,esimon}@cs.ucsd.edu
CSE Department UC, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114

Abstract

This paper studies the relationship between storage requirements and performance. Storage-related dependences inhibit optimizations for locality and parallelism. Techniques such as renaming and array expansion can eliminate all storage-related dependences, but do so at the expense of increased storage. This paper introduces the *universal occupancy vector* (UOV) for loops with a regular stencil of dependences. The UOV provides a schedule-independent storage reuse pattern that introduces no further dependences (other than those implied by true flow dependences). OV-mapped code requires less storage than full array expansion and only slightly more storage than schedule-dependent minimal storage.

We show that determine if a vector is a UOV is NP-complete. However, an easily constructed but possibly non-minimal UOV can be used. We also present a branch and bound algorithm which finds the minimal UOV, while still maintaining a legal UOV at all times.

Our experimental results show that the use of OV-mapped storage, coupled with tiling for locality, achieves better performance than tiling after array expansion, and accommodates larger problem sizes than untilable, storage-optimized code. Furthermore, storage mapping based on the UOV introduces negligible runtime overhead.

1 Introduction

This paper gives a method for space-efficient storage mapping which allows for flexible loop scheduling. The method is applicable to *regular* loops (perfectly nested loops with a regular stencil of data dependences) which produce temporary values that are not used outside the loop. We can determine whether our assumptions are valid for a given loop nest by applying array region analysis [11] and value-based

dependence analysis.

Consider the simple example loop shown in Figure 1. Assume that the zero-th row of the array A , $A[0, *]$, is initialized prior to loop entry, and that the zero-th column contains the same constant value in each entry. Further assume that only the n th row of A is used subsequently in the program. A 's two-dimensional storage of temporary values requires nm storage locations. Our aim is to reduce the amount of storage used, but in a way that does not inhibit other optimizations.

We can graphically represent our simple example with an iteration space graph (ISG) [27]. A loop nest of depth k is represented by a k -dimensional ISG, where the indices in the j th dimension are the values taken on by the j th innermost index variable. Additional iteration points (circles in Figure 1(a)) are added to the ISG to represent the initial values used in the loop. Directed edges represent value dependences from an assignment in one iteration to a use in another iteration.

In the ISG in Figure 1(a) we note that once iteration (i, j) has computed the new value for $A[i, j]$, all uses of $A[i-1, j-1]$, the value produced in iteration $(i-1, j-1)$, have already been consumed in *any* legally scheduled execution. Therefore, we can store the value produced by iteration (i, j) in the same storage location used by iteration $(i-1, j-1)$. We call the relationship between (i, j) and $(i-1, j-1)$ an occupancy vector (OV). If, as in this case, the OV can be used with *any* legal schedule, it is called a *universal occupancy vector* (UOV). Once we have chosen a UOV, we can reduce the amount of storage in our simple example from mn to $n + m + 1$, as shown in Figure 1(b).

It is possible to reduce the storage requirements for this program to $m+2$. For example, we could rewrite it as shown in Figure 1(c). For simplicity the figure shows only some of the storage-related dependences¹. Actually because of the assignment to `temp2` there are storage-related dependences between all iterations. Because of these storage-related dependences, the schedule of the loop is severely restricted. Consequently, data locality optimizations like tiling [15, 26] and loop interchange [27] are not possible without introducing more storage, because they alter the loop schedule. These optimizations are important for achieving good performance[27]. In this paper, we focus on the applicability of tiling, and we give performance results using our techniques in Section 5.

So far we've illustrated how using the UOV reduces storage compared to the use of two-dimensional arrays, and

¹also known as anti- and output dependences [27]

Copyright (c) 1998 by the Association for Computing Machinery, Inc. Permission to make digital/hard copy of all or part of this material without fee is granted provided that copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

This paper appeared in the Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1998.

allows more flexibility for optimization than the storage-optimized version. There is a second scenario where the UOV can help. Suppose we had started with the code in Figure 1(c). We could first perform array and scalar expansion [12, 3] or find the natural subspaces [16] of the variables in the loop in order to remove the storage-related dependences that inhibit optimizations. For instance, in our example, natural subspace expansion would require $3mn$ storage locations. Not only would we be storing all temporary results, we would also be storing each result in three different locations. However, if we next perform forward substitution [3], each value is stored in one location, and we obtain the original code in Figure 1(a). Thus we see that using storage expansion techniques on storage-optimized code presents an opportunity to apply our techniques, by removing storage-related dependences. The final result uses $m + n + 1$ storage — more than the storage-optimized code — but enables later optimizations.

In this paper we define and show how to compute *universal occupancy vectors* which allow data locality transformations while keeping temporary storage requirements minimal. In Section 2 we discuss the program analysis and storage transformations that we use. In Section 3 we describe how to find the universal occupancy vector for a regular loop of any dimensionality. Section 4 discusses the implementation of an OV-based storage mapping in the two dimensional case. In Section 5 we give experimental results that show the OV-based storage mappings introduce negligible overhead, and when used in conjunction with the tiling transformation, can help improve performance. In Section 6 we describe related work, and in Section 7 we draw conclusions.

2 Background

We handle loops whose ISGs contain data dependences with constant distances. We assume that each ISG node (except for the input and output nodes) has the same pattern of dependences, which is called a *stencil* [23]. Also, this technique is only applicable in loops which generate temporary values.

Value-based dependence analysis is fundamental to our work, as it allows us to summarize which iteration produced the values used by each iteration in the ISG. Precise value-based dependence analysis was first developed in [13] over a restricted domain of structured programs, and extended in [20] to obtain the same precision in common cases with greater efficiency. This work uses *Last Write Trees* to represent a mapping from a node in the ISG where a read occurs to another node where the last instance of the value used by the read is written. The work in [21] uses the Omega system to obtain value-based dependence analysis, again with greater efficiency than [13].

The array region analysis of [11] allows us to determine which elements in an array are being imported to the loop and which elements are being exported. Using this information we can determine what storage is used for input and output versus temporary results.

Data locality loop transformations [25, 6] such as tiling change the schedule of a loop to improve the data locality within the loop which in turn can increase performance. Such data locality loop transformations use data dependence information to determine legality. The presence of storage-related data dependences restrict their applicability. Therefore, we use techniques to remove storage-related dependences [12, 3, 16, 5]. Later, when we map storage for reuse, storage-related dependences are reintroduced. How-

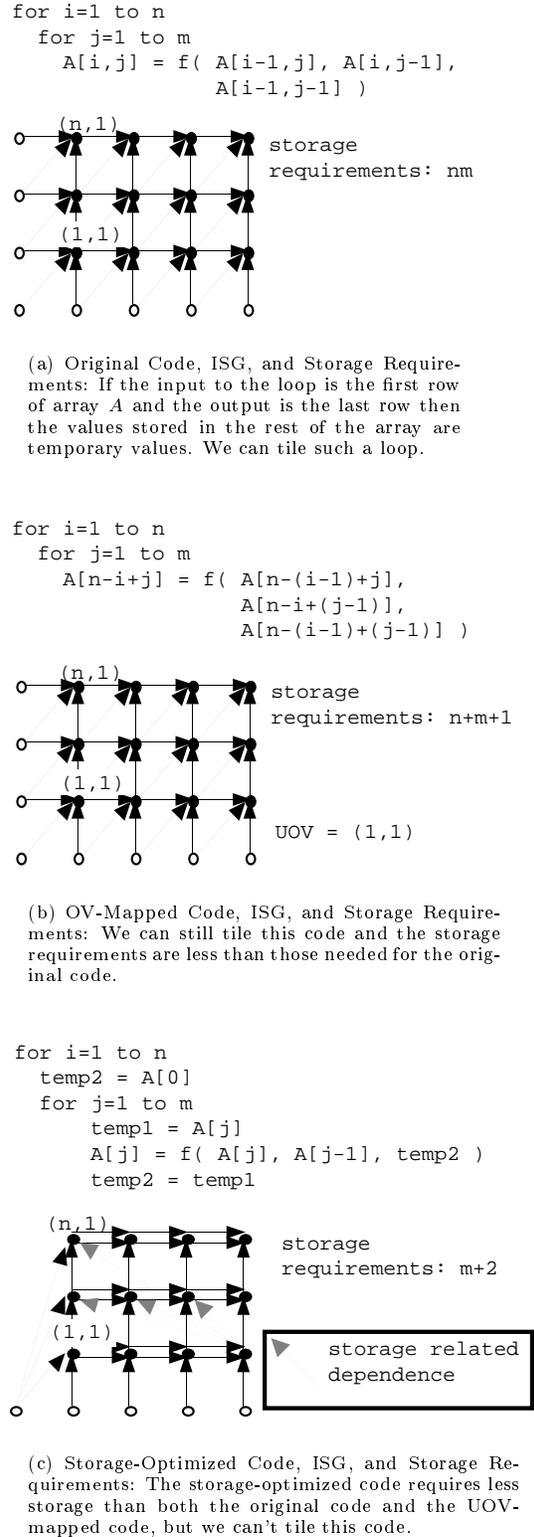


Figure 1: Simple Example

ever, we choose a UOV that only introduces dependences that are in the transitive closure of the stencil of true data dependences, and consequently does not restrict the set of legal schedules.

In this paper we focus on the use of the UOV in tandem with tiling. Tiling [15, 26] is an optimization that partitions the ISG into atomic units of execution called tiles. Tiling changes the execution order of a loop to take better advantage of data locality. It can also be used as a technique to implement parallelism in a loop nest.

Storage mappings based on a universal occupancy vector are especially useful for tiling because of the type of schedules that tiling generates. In tiling, atomic units of the ISG are executed in some order. If there are storage-related dependences between two tiles, then the output of the producer tile has to be saved so that the consumer tile can use it as input. By determining the UOV independent of the schedule we increase the applicability of tiling without doing array expansion.

3 Storage Reuse via the Occupancy Vector

Our technique focuses on one assignment at a time. If the loop has multiple assignments, we would treat each separately, resulting in disjoint storage for the loop-carried values produced by the different assignment statements. We restrict the edges in the ISG to just the edges that correspond to values produced by the assignment under consideration. We call this the *reduced ISG*. Hereafter, we assume that each point in the reduced ISG has the same stencil of data dependences.

We are now ready to determine how the given array or scalar assignment can reuse storage in an efficient but schedule-independent manner. After all iterations that consume the value produced by the assignment in an arbitrary iteration $\vec{p} = (p_1, p_2, \dots, p_d)$ have executed, we no longer need to store that value. Therefore, a later iteration \vec{q} can reuse the space formerly used by iteration \vec{p} . When we reuse storage in this manner, the vector difference $\vec{p} - \vec{q}$ is called the *occupancy vector* (OV), because the data produced by \vec{p} and \vec{q} will occupy the same storage location.

3.1 Universal Occupancy Vectors

Since iteration \vec{q} now stores a value in the same storage location used by iteration \vec{p} , a def-def storage-related dependence between those two iterations has been created. Furthermore, if \vec{k} is an iteration that uses the value defined in \vec{p} , then there is a use-def storage-related dependence between \vec{k} and \vec{p} . In order to keep these dependences from restricting the set of schedules for the loop, we will pick an OV, $\vec{ov} = \vec{q} - \vec{p}$, such that \vec{p} , and also any iteration \vec{k} that uses the values produced at \vec{p} , is guaranteed to have been executed before \vec{q} in any legal schedule. We will call \vec{ov} a *universal occupancy vector* (UOV).

To develop intuition on how to find a universal occupancy vector, we define two sets for a given iteration point \vec{q} with a given stencil $V = \{\vec{v}_1, \dots, \vec{v}_m\}$. The $DONE(V, \vec{q})$ set contains iteration points that must be executed before \vec{q} because of value dependences. In the example shown in Figure 2, the iteration points in the $DONE$ set for the circled point are black.

$$DONE(V, \vec{q}) = \{\vec{p} \mid \exists a_i \geq 0, \vec{p} + \sum a_i \vec{v}_i = \vec{q}\}$$

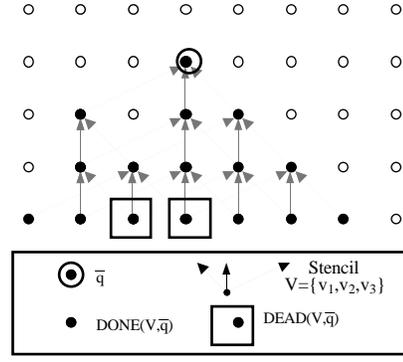


Figure 2: $DONE$ and $DEAD$ set – The $DONE$ set contains points that must be executed before arbitrary point \vec{q} . The $DEAD$ set contains points whose produced values are no longer needed once \vec{q} consumes its inputs. Notice that points in the $DEAD$ set have all three dependences satisfied once \vec{q} is executing.

The set $DEAD(V, \vec{q})$ contains iteration points whose produced values are no longer needed once the iteration \vec{q} has consumed its values. Figure 2 denote points in the $DEAD$ set with squares.

$$DEAD(V, \vec{q}) = \{\vec{p} \mid \forall \vec{v}_i \in V, \vec{p} + \vec{v}_i \in DONE(V, \vec{q})\}$$

Note that $DEAD(V, \vec{q}) \subseteq DONE(V, \vec{q})$. For any point $\vec{p} \in DEAD(V, \vec{q})$, the value produced in iteration \vec{q} can be mapped to the same storage as the value produced in \vec{p} . Thus, the set of legal universal occupancy vectors is

$$UOV(V) = \{\vec{q} - \vec{p} \mid \vec{p} \in DEAD(V, \vec{q})\}$$

Given our assumption that the stencil of dependences is the same at each point, the set $UOV(V)$ does not depend on the choice of \vec{q} .

From the definition of the $DEAD$ and $DONE$ sets, we know that for any universal occupancy vector \vec{ov} , the following equations must have a solution with each coefficient a_{ij} being a non-negative integer, and the diagonal coefficients a_{ii} being positive integers:

$$\begin{aligned} \vec{ov} &= a_{11}\vec{v}_1 + \dots + a_{1m}\vec{v}_m \\ &\dots \\ \vec{ov} &= a_{m1}\vec{v}_1 + \dots + a_{mm}\vec{v}_m \end{aligned}$$

Unfortunately, determining whether a given vector \vec{w} is in $UOV(V)$ is NP-complete. In particular:

Theorem Given a set V of dependences and an arbitrary vector \vec{w} , determining whether \vec{w} is in $UOV(V)$ is NP-complete.

Proof: The proof is via a reduction of the PARTITION problem [14]. Suppose we are given an instance of the partition problem, expressed as a sequence a_0, a_1, \dots, a_{n-1} of positive integers.² Let $h = \sum a_i/2$. The partition problem is to determine if there is a subsequence that has sum h . We construct an instance of the UOV membership problem. The stencil V will be a set of two-dimensional vectors. For each

²We use sequences instead of sets to allow duplicate values, conforming with [14].

a_i , V will include the vectors $r_i^z = (0, (n+1)^i + (n+1)^n)$ and $s_i^z = (a_i, (n+1)^i + (n+1)^n)$. Let $\vec{w} = (h, n(n+1)^n + ((n+1)^n - 1)/n)$. It can be shown that $\vec{w} \in UOV(V)$ if and only if the instance of the partition problem has a solution.³ Furthermore, determining if $\vec{w} \in UOV(V)$ is in NP since a solution to the set of equations shown above can be “guessed” and verified in polynomial time. Thus, the UOV-membership problem is NP-complete. \square

Fortunately, there is a trivially-computed initial UOV which can be calculated by summing the vectors in the set V . Also, we believe our algorithm for finding the best UOV will be efficient for realistic problem instances.

3.2 Finding the Optimal Universal Occupancy Vector

An occupancy vector partitions the iteration points in the reduced ISG into storage-equivalent classes. Any two points are storage-equivalent if they differ by an integral multiple of the OV. The iteration points in each class can use the same locations to store the values they produce for the array or scalar in question. The storage determined by an OV is therefore the number of such classes times the storage needed for the defs of the array or scalar. An *optimal* universal occupancy vector is one that requires the least amount of storage while still allowing for any legal loop schedule.

Given $\vec{o}\vec{v}$, the number of storage-equivalent classes is the number of integer points in the volume described by $\vec{o}\vec{v}$ and the projection of the ISG on the hyperplane perpendicular to $\vec{o}\vec{v}$. Typically the bounds of an ISG are not known until runtime because they are expressed as variables. In this case the best way to reduce the storage requirements is to find the shortest OV. In general however, a longer OV can require less storage depending upon the projection of the ISG. For example, in Figure 3 of the two OVs, $\vec{o}\vec{v}_1 = (3, 1)$ and $\vec{o}\vec{v}_2 = (3, 0)$, $\vec{o}\vec{v}_2$ is obviously shorter; however, because of the angle and size of the ISG $\vec{o}\vec{v}_1$ requires less storage. Specifically the projection of the ISG on the hyperplane perpendicular to $\vec{o}\vec{v}_1$ is small enough to offset the difference in length between $\vec{o}\vec{v}_1$ and $\vec{o}\vec{v}_2$. Therefore, if the bounds of the ISG are known at compile time, their projection must be considered in order to find the best OV.

Using the volume described by $\vec{o}\vec{v}$ and the projection of the ISG on the perpendicular hyperplane, we can compare different UOVs by the amount of storage they require. We will describe an exhaustive search procedure that should efficiently find the optimal UOV in practice.

3.2.1 Bounding the Search Area

An initial UOV can be trivially computed by summing the value dependences in the stencil, $\vec{o}\vec{v}_o = \sum_{i=1}^m \vec{v}_i$. This allows us to bound the area in which to search for the optimal UOV. We need only search the subset of *DEAD* that could give us an occupancy vector that requires less storage than the initial estimate $\vec{o}\vec{v}_o$.

Assume the dimensions of the ISG are unknown, so our goal is to find the shortest OV. Therefore, we only need to check the points of the ISG inside a sphere to see if they are in *DEAD*, where the starting bound on the sphere’s radius is the length of the initial occupancy vector $\vec{o}\vec{v}_o$. To do so, it is necessary to examine points in a larger area to see if they are members of *DONE* – those that may be single stencil

³The magic numbers in the second coordinates of the vectors ensures that if \vec{w} is expressed as a sum of elements of V , then there must be exactly n vectors in the sum and they must include exactly one of r_i^z and s_i^z for each i . The a_i ’s corresponding to the s_i^z terms provide a solution to the partition problem.

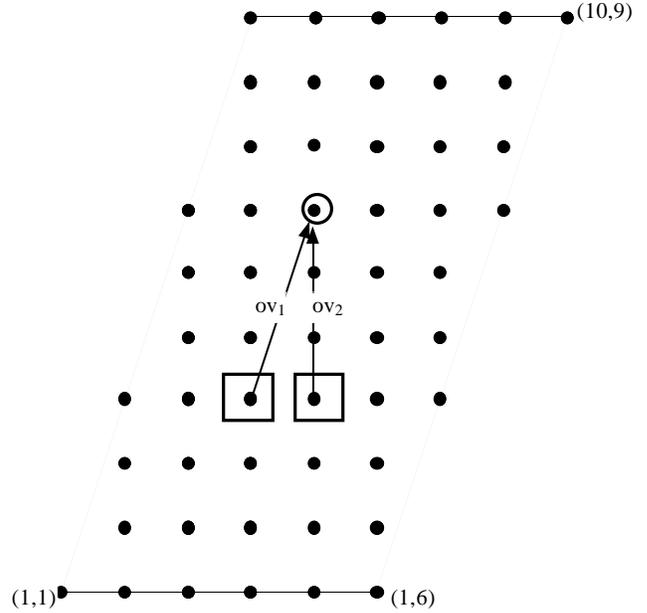


Figure 3: ISG with constant bounds – If the ISG for the stencil in Figure 2 was the size shown above the shortest possible OV requires more storage than a slightly longer OV. $\vec{o}\vec{v}_2$ requires 27 storage locations while $\vec{o}\vec{v}_1$ only requires 16 storage locations.

dependence away from candidate points in the sphere. In Figure 4 we show how the search would be bound in a two-dimensional ISG. We use the extreme vectors [22] from the stencil to create a parallelepiped for the search for points in *DONE*. This parallelepiped contains all points that are a single stencil dependence away from candidate points in the sphere.

At any point during the traversal if we find an iteration point in the *DEAD* set which gives an occupancy vector with length less than the bound, we reset the bound to this new value.

In the case where we know the size of the ISG at compile time, we must take the projection of the ISG on the hyperplane perpendicular to the UOV into account when determining which UOV requires the least amount of storage. Let P_M denote the minimum projection of the ISG on *any* hyperplane.⁴ Let $\vec{o}\vec{v}_o = \sum_{i=1}^m \vec{v}_i$ again be our initial occupancy vector, and let $P_{\vec{o}\vec{v}_o}$ be the projection of the ISG on the hyperplane perpendicular to $\vec{o}\vec{v}_o$. The amount of storage needed if the initial UOV $\vec{o}\vec{v}_o$ is chosen is the number of integer points in the volume $P_{\vec{o}\vec{v}_o}|\vec{o}\vec{v}_o|$. Therefore the best occupancy vector $\vec{o}\vec{v}_{best}$ would satisfy the following inequality $P_{\vec{o}\vec{v}_o}|\vec{o}\vec{v}_o| \geq P_M|\vec{o}\vec{v}_{best}|$. The search bound put on the length of the best occupancy vector is $P_{\vec{o}\vec{v}_o}|\vec{o}\vec{v}_o|/P_M$.

3.2.2 Branch and Bound Search

Given an initial bound we can do a branch and bound search. Since the initial bound is based on a legal UOV, from the very start of the algorithm we have a possible solution. However, this solution might be far worse than the best solution.

⁴For example, in the case of a rectangle, this corresponds to the side with the shortest length.

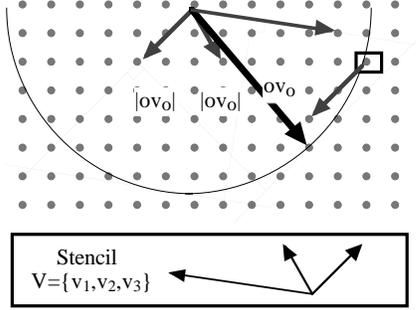


Figure 4: Bounding the Search – $o\vec{v}_0$ is our initial occupancy vector determined by summing all of the dependences in the stencil. If $|\vec{ov}_0|$ is the bound, our search space is the parallelogram shown.

A compiler could limit the amount of time the algorithm runs and just take the best answer found so far.

The search starts at an arbitrary point in the reduced ISG, \vec{q} . A breadth-first like search of the reversed value dependences allows us to discover which iterations are in the $DONE(V, \vec{q})$ set. The search begins by inserting an arbitrary starting point \vec{q} into a priority queue. The priority is based either on the point's distance from \vec{q} or the calculation of the storage required if the size of the ISG is known. The best UOV will have a lower priority than other candidates so by using a priority queue we insure that the best candidates are examined first. Consequently, we expect the best answer to be found quickly.

Each time an iteration point \vec{p} is visited (i.e. extracted from the priority queue) it is examined to see if it is a better UOV than the one currently bounding the search space. A point is in $UOV(V)$ if all of the value dependences in V have been traversed at least once to reach this point from \vec{q} . To keep track of which dependences have been traversed each point has a $PATHSET$. Anytime $PATHSET = V$ the point is a legal UOV and determines a new bound if it is smaller than the current bound. Also while visiting a point, the point's children (i.e. points which can be reached by traversing the backward value dependences) are inserted into the priority queue with their $PATHSET$'s updated. We summarize the algorithm here.

Algorithm VISIT(\vec{p})

1. Check that point \vec{p} is within boundaries, if not then return.
2. For each $child_k$ (the child reached by following value dependence $\vec{v}_k \in V$), if the parent's $PATHSET$ contains more dependences than the child's $PATHSET$ or if \vec{v}_k is not already in the child's $PATHSET$, then update the child's $PATHSET$ with these dependences and put the child into the priority queue.
3. If $(\vec{q} - \vec{p}) \in UOV(V)$ then check whether the derived UOV gives a better bound than the current bound.

The worst-case running time of the overall algorithm is the number of iteration points within the initial bounds.

4 Determining Storage Mappings

After selecting an occupancy vector, universal or otherwise, we must determine a storage mapping in order to generate code. In this section we describe some general requirements for OV-based storage mapping and give the details in the two-dimensional case.

A storage mapping is a function which, given an iteration \vec{q} , returns an integer index into one-dimensional memory. Typically a d -dimensional array is mapped into one-dimensional memory in either column-major order

$$(q_1, q_2, \dots, q_d) \rightarrow q_1 + (s_1)q_2 + (s_1s_2)q_3 + \dots + (s_1 \dots s_{d-1})q_d$$

or row-major order

$$(q_1, q_2, \dots, q_d) \rightarrow q_1(s_2 \dots s_d) + q_2(s_3 \dots s_d) + \dots + q_d$$

where s_1, s_2, \dots, s_d represent the sizes of each array dimension. Both of these mappings are equivalent to taking the dot product of the iteration point with a vector of constants

$$(q_1, q_2, \dots, q_d) \cdot (1, (s_1), (s_1s_2), \dots, (s_1 \dots s_{d-1}))$$

$$(q_1, q_2, \dots, q_d) \cdot ((s_2s_3 \dots s_d), (s_3s_4 \dots s_d), \dots, (s_d), 1)$$

The amount of work done to map a d -dimensional array is $(d-1)$ multiplies and $(d-1)$ additions. Optimally, OV-based storage mappings should have no more overhead than standard array mappings. Therefore, we derive a mapping vector \vec{mv} which will result in an integer index into one-dimensional storage when the dot product with an iteration is taken, $\vec{q} \cdot \vec{mv}$.

The storage mapping is of the form:

$$SM_{ov}(\vec{q}) = \vec{mv} \cdot \vec{q} + shift + modterm$$

Here, $shift$ is simply a constant term that ensures that the result returned by the function is always non-negative. The mapping vector maps the iterations into relative locations in one-dimensional storage. The modterm deals with *non-prime*⁵ OVs. The storage mapping requires up to d multiplies and $(d+1)$ additions where d is the dimensionality of the ISG. In the worst case it requires one more multiply and two more adds than usual array indexing. However, the number of multiplies required depends upon the actual value of the elements in \vec{mv} and whether there is a $shift$ and/or $modterm$. In the simple example shown in Figure 1(b) we use the storage mapping

$$SM_{ov}(\vec{q}) = (-1, 1) \cdot \vec{q} + n$$

The mapping requires only one subtraction and one addition instead of the two multiplies and four additions which might be required in the general two-dimensional case.

Note that since we are taking complete control of temporary storage allocation, it would not be difficult to incorporate data layout techniques such as array padding [3] to improve performance. In addition, the techniques of [2] may be of further use in improving our storage mapping. That work first parallelizes sequential programs to minimize communication, then changes the data layout (using strip-mining and permutation) to ensure contiguous storage and take advantage of locality on each processor. We could use their techniques to improve data layout once we have chosen an occupancy vector.

Next we will discuss the details of calculating the mapping vector \vec{mv} , the $modterm$, and the amount of storage to allocate.

⁵ ov is non-prime if it passes through points in the ISG other than the head and tail.

4.1 Mapping Vector

The mapping vector must satisfy the following requirements given an occupancy vector \vec{ov} :

1. If $\vec{q} - \vec{p} = \vec{ov}$ then \vec{q} and \vec{p} must map to the same storage location.
2. Each iteration must map to an integer location.
3. To efficiently use storage, consecutive storage locations should be used.

In two-dimensions, first suppose that $\vec{ov} = (i, j)$ is *prime*, meaning that $\text{GCD}(i, j) = 1$. In this case, the mapping vector \vec{mv} can simply be chosen as $(-j, i)$. The first requirement is satisfied because if two vectors are \vec{ov} apart in the ISG their storage locations will be $(i, j) \cdot (-j, i) = 0$ apart. Secondly, since \vec{ov} has integer elements, the calculated mapping vector will also have integer elements. Finally, by the Euclidean algorithm there exists integers a and b such that $ai + bj = \text{GCD}(i, j)$. Therefore, storage locations will be consecutive.

4.2 Non-Prime Occupancy Vectors

modterm deals with the case that \vec{ov} passes through multiple points in the ISG. The \vec{ov} semantics demand that the same storage location be used for any ISG points that are \vec{ov} apart. However, because of our mapping method, any ISG points that lie along \vec{ov} will be mapped to the same storage location. In two dimensions, the number of different storage-equivalent classes within a single $\vec{ov} = (i, j)$ is the greatest common denominator, $\text{GCD}(i, j)$. We can use the *modterm* to determine which of the $\text{GCD}(i, j)$ classes a ISG point falls into.

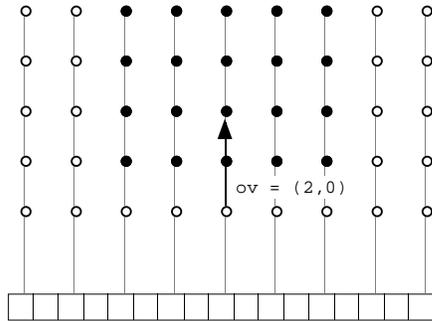


Figure 5: The UOV for our 5-point stencil code intersections two integer points. This code is discussed in Section 5. Here we depict interleaved storage.

There are two different ways we can lay out storage under these circumstances. Either we can layout all of the storage for one modclass followed by all of the storage for the next modclass, etc. or we can interleave the storage for the various modclasses. If the storage is interleaved then the mapping vector \vec{mv} is chosen as described above. For the example in Figure 5 $\vec{mv} = (0, 2)$. All three conditions of the mapping vector are still met except that now the points being mapped to are $\text{GCD}(i, j)$ apart. The *modterm* will be used to fill in these extra positions. For example, in Figure 5 our storage mapping would be as follows:

$$SM_{ov}(\vec{q}) = (0, 2) \cdot \vec{q} + (q_1 \bmod 2)$$

If the storage not interleaved then the mapping vector must be divided by $\text{GCD}(i, j)$ so that consecutive memory locations are used. The *modterm* will then be used to select a consecutive group of storage for the iteration point being considered. In this case, in Figure 5 the storage mapping would be

$$SM_{ov}(\vec{q}) = (0, 1) \cdot \vec{q} + (q_1 \bmod 2) * L$$

where L is the length of the ISG projection on the line perpendicular to \vec{ov} . In generating code, we remove the overhead introduced by the *mod* operations by applying loop unrolling [3].

4.3 Storage Allocation

We will compute the number of storage-equivalent classes as follows. We first compute a mapping vector \vec{mv} , as described in Section 4.1, which ensures that the iteration points projected along \vec{ov} map to integer points. Once we have computed the mapping vector, we apply it to the extreme points of the ISG⁶, $\vec{x}\vec{p}_1$ and $\vec{x}\vec{p}_2$, obtaining the number of integer points in this projection. If the OV is non-prime the number of storage-equivalent classes which lie along the OV must be taken into account. See Figure 6 for an example of storage allocation computation.

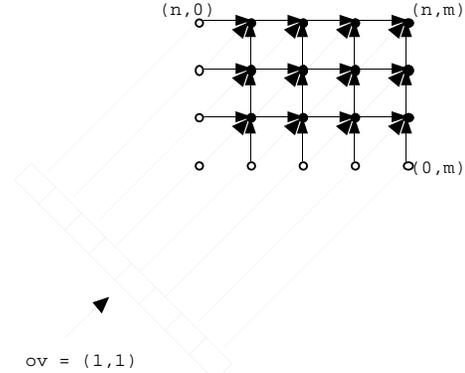


Figure 6: Calculating storage requirements — number of storage locations = $|\vec{mv} \cdot \vec{x}\vec{p}_1 - \vec{mv} \cdot \vec{x}\vec{p}_2| + 1 = |(-1, 1) \cdot (0, n) - (-1, 1) \cdot (m, 0)| + 1 = |n + m| + 1$

5 Experimental Results

We examined the performance of two codes over a range of problem sizes. For each code, we implemented natural, OV-mapped, and storage optimized versions, and also made tiled versions of these codes whenever possible. The natural versions of the algorithms use array expansion. We refer to such code as natural because it is the most natural way for a programmer to write code for the given algorithm (i.e. not worrying about storage issues). The OV-mapped versions use a universal occupancy vector to determine the storage mapping. Finally, the storage optimized versions use the least amount of storage possible given a specific schedule.

⁶The ISG is the set of integer solutions to a system of linear inequalities defined by the loop bounds, $Ai \leq b$. The extreme points are the vertices of this convex polyhedron.

First we compare the performance on problem sizes which fit in cache. This gives us a basis to judge when performance degrades due to memory hierarchy effects. These experiments also allow us to compare the relative overhead costs. We then look at a large range of problem sizes observing when the various versions of the code fall out of memory and also observing whether tiled OV-mapped code provides scaling for very large problem sizes.

Our two codes are implementations of a 5-point one-dimensional stencil and a protein string matching algorithm in C. We ran it on a Pentium Pro running at 200MHz, a 200MHz Sun Ultra 2, and a 500MHz DEC Alpha 21164. The gcc compiler was used on each machine with optimization level 2.

In the 5-point stencil code the values of a 1-D array of length L change over T time steps by taking a weighted average of an element's neighbors. In the natural code version, a two-dimensional array of size TL is used to store all intermediate values in the loop (see Table 1). Both this version and the OV-mapped version use the 1-D input array when calculating the first row of temporary values, and put the last row of results into the output array. This makes it possible to use temporary storage for a loop computation while not having to change code outside the loop. The storage optimized version of 5-point stencil takes advantage of the fact that the output of the loop is just the transformed input; that is, none of the intermediate values are needed upon exit of the loop. The input and output are a 1-D array. The loop uses three temporary scalars to satisfy the data dependences.

The 5-point stencil has the UOV shown in Figure 5. The amount of storage needed to implement this UOV is equivalent to two one-dimensional arrays of size L . The two different OV-mapped versions store the two rows of storage consecutively in memory or interleave the storage. Experiments were done on both storage layouts to compare their performance. Theoretically the interleaved storage will not have associativity problems, but since the references are not consecutive hardware prefetching may not occur.

	Temporary Storage
Natural	TL
OV-Mapped	$2L$
Storage Optimized	$L + 3$

Table 1: 5-point stencil – L is the length of the array being transformed and T is the number of time steps.

The protein string matching code compares two strings of length n_0 and n_1 for similarity. The strings consist of characters representing amino acids. There is storage for the strings themselves and a 23x23 table which holds comparison weights for the 23 possible string characters. The computation compares each character of one string with all of the characters in the other string. The storage optimized version was taken from [1]. See Table 2, for the storage requirements of the different versions.

	Temporary Storage
Natural	$n_0 n_1 + n_1 + n_0$
OV-Mapped	$2n_0 + 2n_1 + 1$
Storage Optimized	$2n_0 + 3$

Table 2: Protein String Matching – This algorithm evaluates the similarity of two strings with lengths n_0 and n_1 .

5.1 Overhead

As discussed in Section 4, OV-based storage-mappings like regular array references require a certain amount of array indexing overhead. The storage optimized version of the code has less array indexing overhead because it uses arrays with fewer dimensions; however unlike the other versions, it has overhead in the form of copies between temporary scalars. We show the relative overheads by comparing the performance of small problem sizes.

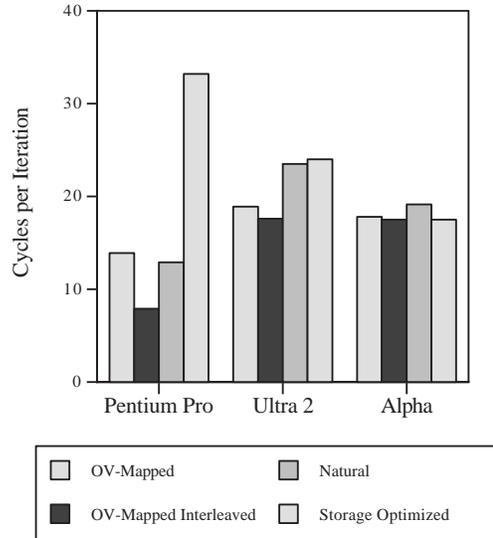


Figure 7: Overhead in 5-point stencil – With problem sizes which fit into L1 cache the various versions of the code have similar performance.

Figure 7 shows the average number of cycles per iteration on problem sizes which fit in cache for the 5-point stencil code. On the Ultra 2 and the Alpha, the different versions of the code perform similarly. On the Pentium Pro there is more variance. The Pentium Pro is a more complicated architecture, and we conjecture that gcc did not consistently take advantage of its microarchitecture.

Figure 8 shows the average number of cycles per iteration for protein string matching. Notice that the OV-mapped codes have relatively less overhead than the natural version of this code. However, the storage optimized version has the lowest relative overhead.

5.2 Scaling to Large Sizes

We now explore how each version of the two codes scales to larger problem sizes. Because the storage-optimized versions of the code use much less storage, they will fall out of cache, TLB, and eventually memory on larger problem sizes. In the performance results, the number of cycles per iteration skyrockets when a version of the code falls out of memory. OV-mapped codes fall out of memory at smaller problem sizes than storage mapped codes, but at much larger problem sizes than natural codes. The natural code and OV-mapped code have the advantage that they can be tiled

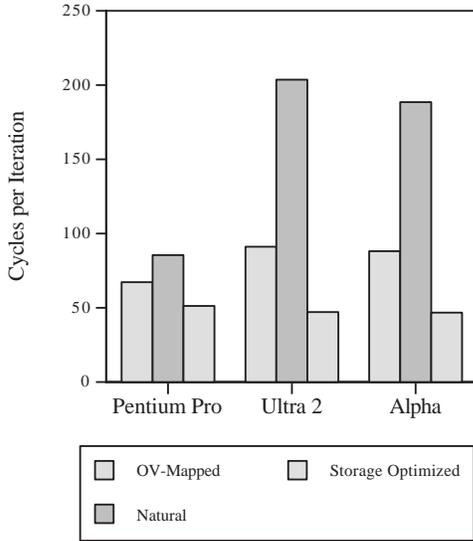


Figure 8: Overhead in Protein String Matching – The performance differs more in the PSM example, OV-Mapped code still does better than the natural version.

for data locality. In all of the following experiments we tiled for L1 cache.

In the 5-point stencil algorithm, tiling in conjunction with an OV-based storage mapping did maintain better performance for very large problem sizes; see Figures 9, 10, and 11. However, tiling the natural codes did not help maintain performance. This is probably due to the fact that the natural version only references each storage location at most twice within a tile, whereas the OV-mapped code references a storage location up to H times, where H is the height of the tile.

Figures 12, 13, and 14 show the performance results for protein string matching. The tiled, OV-mapped protein string matching code had better performance than all other versions of the code on the Pentium Pro, but did not help the scalability of the algorithm on the Ultra 2 and the Alpha. The inner loop of protein string matching has many branches so we conjecture that on the Ultra 2 and the Alpha pipeline stalls due to branches are the bottleneck instead of memory latency.

From these results we conclude that in codes where the memory latency is the bottleneck, OV-mapped codes help performance by allowing tiling while at the same time keeping the storage minimal.

6 Related Work

The most closely related work to ours is [18], which also determines storage reuse for a loop. Their work takes as input a given parallel schedule, but allows direction vectors. In contrast, our UOV-based approach can be used for *any* legal schedule in the context of a loop with constant distance vectors. This allows the flexibility of performing tiling *after* storage mapping.

Other related work is in storage expansion and optimiza-

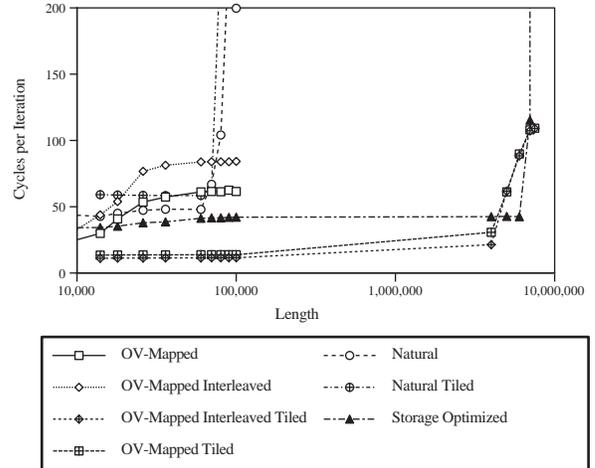


Figure 9: 5-point stencil on the Pentium Pro

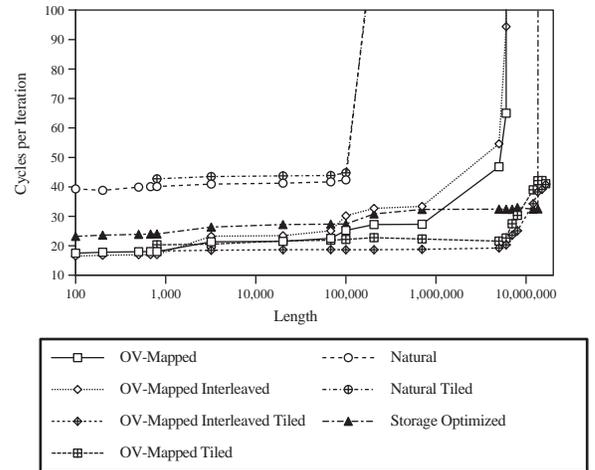


Figure 10: 5-point stencil on the Ultra 2

tion. Array privatization [24, 20] creates separate, per processor storage in a parallel context. In our work, storage expansion is performed on a per iteration basis for better locality. The work in [4] creates a maximal expansion of storage that does not require ϕ -functions, trading off parallelism for memory usage. The goal of our work is data locality. The work in [17] introduces an Array SSA form that keeps track of values on an element-wise basis. Both of these latter works may be useful as we extend our work to more general program structures.

In [9, 10], a unified framework and heuristics that consider the combined effect of array layout and loop transformations (described by integer, non-singular matrices [19]) on locality are developed. More specifically, they seek a legal data and control transformation that will result in a *stride vector* with larger strides at outer loops, and smallest strides

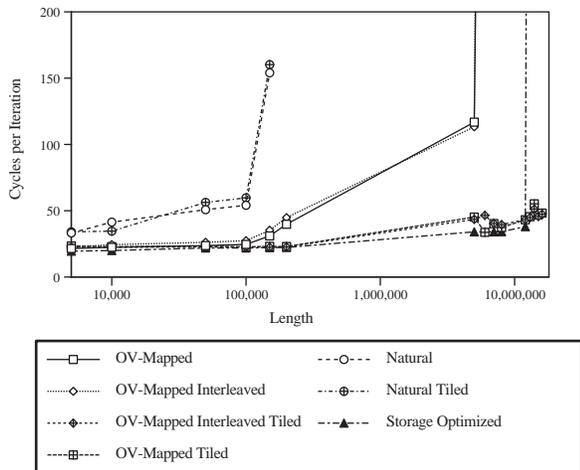


Figure 11: 5-point stencil on the Alpha

at inner loops. The control transformations considered do not include tiling or scheduling.

7 Conclusions and Future Work

This paper presents the universal occupancy vector as a method for reducing the storage requirements of a program without introducing any data dependences which are not already implied by the value dependences. We show that the UOV-membership problem is NP-complete, where the problem size is the number of dependences in the stencil. An algorithm for determining the best UOV is presented. Since the number of dependences is small in practice, our branch and bound algorithm is practical. We give evidence that basing storage reuse on the occupancy vector helps improve performance while reducing storage requirements.

An interesting potential benefit of having the compiler select a UOV for the code is that programmers are encouraged to write more natural codes, and let the compiler deal with determining storage reuse. This allows for code which is easier to write, read and maintain.

Future work will extend the UOV approach to multiple loop nests. We might want to select our occupancy vector in a way that allows two loops to use the same OV-mapping for a given array.

In this paper we gave evidence showing one level of tiling along with use of the occupancy vector improves performance. We plan to study which characteristics of the entire memory hierarchy should be taken into account when doing multiple-level optimizations like hierarchical tiling[7, 8].

8 Acknowledgement

We would like to thank Ken Lyons at AT&T Labs and Nick Mitchell and Kang Su Gatlin of UCSD for their helpful comments and suggestions. This work is partly supported by NSF grant CCR-9504150, a NSF graduate research fellowship, and an AT&T Labs student grant.

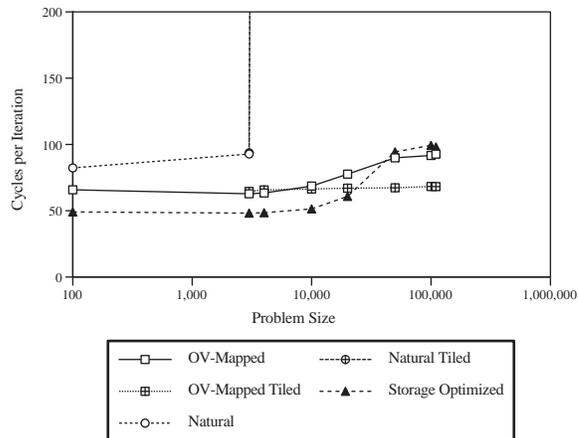


Figure 12: Protein String Matching on the Pentium Pro. Natural Tiled coincides with Natural and therefore can't be seen on graph.

References

- [1] B. Alpern, L. Carter, and K.S. Gatlin. Microparallelism and high-performance protein matching. In *Supercomputing*, 1995.
- [2] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica Lam. Data and computation transformations for multiprocessors. In *SIGPLAN PLDI*, La Jolla, CA, June 1995.
- [3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *Computing Surveys*, 26(4):345-420, 1994.
- [4] Denis Barthou, Albert Cohen, and Jean-François Colliard. Maximal static expansion. In *Principles of Programming Languages*, January 1998.
- [5] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1-2):251-266, 1997.
- [6] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *ASPL OS VI*, pages 252-262, San Jose, CA, November 1994.
- [7] Larry Carter, Jeanne Ferrante, and S. Flynn Hummel. Efficient parallelism via hierarchical tiling. In *Proceedings of SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
- [8] Larry Carter, Jeanne Ferrante, Susan Flynn Hummel, Bowen Alpern, and Kang Su Gatlin. Hierarchical tiling: A methodology for high performance. Technical Report CS96-508, UCSD, Department of Computer Science and Engineering, November 1996.
- [9] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *SIGPLAN PLDI*, La Jolla, CA, June 1995.

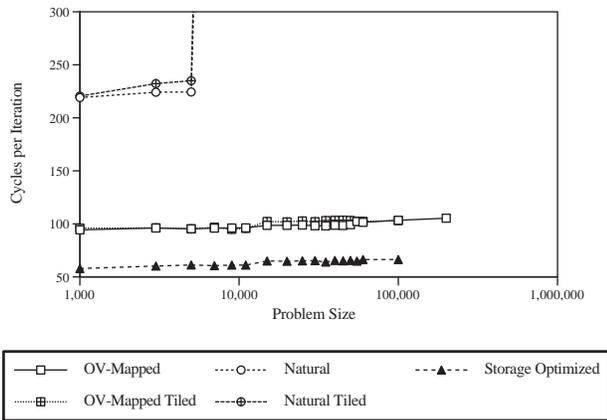


Figure 13: Protein String Matching on the Ultra 2

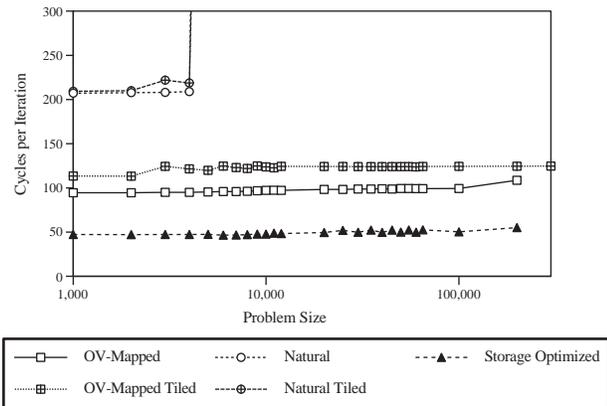


Figure 14: Protein String Matching on the Alpha

[10] Michal Cierniak. *Optimizing Programs by Data and Control Transformations*. Ph.d. thesis, Rochester, Computer Science Department TR670, November 1997.

[11] Béatrice Creusillet and François Irigoien. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, December 1996.

[12] P. Feautrier. Array expansion. In *International Conference on Supercomputing*, pages 429–442, 1988.

[13] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1), February 1991.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. Bell Telephone Laboratories, Incorporated, 1979.

[15] F. Irigoien and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN*

Symposium on Principles of Programming Languages, pages 319–329, 1988.

- [16] Kathleen Knobe and William J. Dally. The subspace model: A theory of shapes for parallel systems. In *5th Workshop on Compilers for Parallel Computers*, Malaga, Spain, 1995.
- [17] Kathleen Knobe and Vivek Sarkar. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, January 1998.
- [18] V. Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Journal on Parallel Computing*, 1997. To be published.
- [19] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal on Parallel Processing*, 22(2), April 1994.
- [20] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, January 1993.
- [21] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.
- [22] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings Supercomputing 91*, pages 111–120, 1991.
- [23] Daniel A. Reed and Loyce M. Adams and Merrell L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.
- [24] Peng Tu and David Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computing 6th International Workshop*, pages 500–521, August 1993.
- [25] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.
- [26] Michael J. Wolfe. Iteration space tiling for memory hierarchies. In *Parallel Processing for Scientific Computing*, pages 357–361, 1987.
- [27] Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.