Array Unification: A Locality Optimization Technique

Mahmut Taylan Kandemir

Computer Science and Engineering Department
The Pennsylvania State University
University Park, PA 16802-6106, USA
E-mail: kandemir@cse.psu.edu
WWW: http://www.cse.psu.edu/~kandemir

Abstract. One of the key challenges facing computer architects and compiler writers is the increasing discrepancy between processor cycle times and main memory access times. To alleviate this problem for a class of array-dominated codes, compilers may employ either control-centric transformations that change data access patterns of nested loops or data-centric transformations that modify the memory layouts of multi-dimensional arrays. Most of the layout optimizations proposed so far either modify the layout of each array independently or are based on explicit data reorganizations at runtime.
This paper describes a compiler technique, called *array unification,* that automatically maps multiple arrays into a single data (array) space to improve data locality. We present a mathematical framework that enables us to systematically derive suitable mappings for a given program. The framework divides the arrays accessed by the program into several groups and each group is transformed to improve spatial locality and reduce the number of conflict misses. As compared to the previous approaches, the proposed technique works on a larger scope and makes use of independent layout transformations as well whenever necessary. Preliminary results on two benchmark codes show significant improvements in cache miss rates and execution time.

1 Introduction

Processor cycle time continues to decrease at a much faster rate than main memory access times, making the cache memory hierarchy performance critical. To address this issue, conventional cache management techniques must be supported by software-oriented approaches. Throughout the years, several compiler techniques have been proposed and implemented with the objective of making memory access patterns of applications more cache-friendly. These include modifications to control structures (e.g., nested loops), careful placement of load/store operations, and reformatting memory layouts for scalar, multi-field (e.g., records), and array variables.

Modifications to control structures and smart load/store placement techniques are known to be bounded by inherent data dependences in the code.

Reformatting memory layouts, on the other hand, is less problematic as it does not change the data dependence structure of the computation. In particular, recent years have witnessed a vast group of layout optimization techniques that target at codes with inherent data locality problems such as regular numerical codes that manipulate large multi-dimensional arrays [11,9] or codes that utilize pointer structures (e.g., linked lists and trees) [1].

Previous research on data transformations for numerical codes focussed primarily on transforming a single array at a time. This special case is interesting mainly because it may enable improved spatial locality for each transformed array. For instance, the studies presented by Leung and Zahorjan [11], Cierniak and Li [2], and Kandemir et al. [9] fall into this category. Most of these transformations, like many loop-oriented optimization techniques, target only the reduction of capacity misses, namely the misses that are due to small cache sizes. However, in particular, in caches with low associativity, conflict misses can present a major obstacle to realizing good cache locality even for the codes that have specifically been optimized for data locality, thereby precluding effective cache utilization. On top of this, as opposed to capacity misses, the degree of conflict misses can vary greatly with slight variations in array base addresses, problem (input) sizes, and cache line (block) sizes [17].

In this paper, we discuss a data space transformation technique, which we call *array unification*, that transforms a number of array variables simultaneously. It achieves this by mapping a set of arrays into a common array space and replacing all references to the arrays in this set by new references to the new array space. One of the objectives of this optimization is to eliminate inter-variable conflict misses, that is, the conflict misses that are due to different array variables. Specifically, we make the following contributions:

- We present a framework that enables an optimizing compiler to map multiple array variables into a single data space. This is the *mechanism aspect* of our approach and leverages off the work done in previous research on data (memory layout) transformations.
- We present a global strategy that decides which arrays in a given code should be mapped into common data space. This is the *policy aspect* of our approach and is based on determining the temporal relations between array variables.
- We present experimental results showing that the proposed approach is successful in reducing the number of misses and give experimental data showing the execution time benefits.
- We compare the proposed approach to previous locality optimizations both from control-centric domain [19] and data-centric domain [11,9].

The rest of this paper is organized as follows. Section 2 gives the background on representation of nested loops, array variables, and data transformations and sets the scope for our research. Section 3 presents the array unification in detail. Section 4 introduces our experimental platform and the benchmark codes tested, and presents performance numbers. Section 5 discusses related work and Section 6 summarizes the contributions and gives an outline of future work.

2 Background

Our focus is on affine programs that are composed of scalar and array assignments. Data structures in the program are restricted to be multidimensional arrays and control structures are limited to sequencing and nested loops. Loop nest bounds and array subscript functions are affine functions of enclosing loop indices and constant parameters. Each iteration of the nested loop is represented by an *iteration vector,* $\boldsymbol{I'}$, which contains the values of the loop indices from outermost position to innermost. Each array reference to an m-dimensional array U in a nested loop that contains n loops (i.e., an n-level nested loop) is represented by $R_u\boldsymbol{I}+\boldsymbol{o_u}$, where $\boldsymbol{I}$ is a vector that contains loop indices. For a specific $\boldsymbol{I} = \boldsymbol{I'}$, the data element $R_u\boldsymbol{I'} + \boldsymbol{o_u}$ is accessed. In the remainder of this paper, we will write such a reference as a pair $\{R_u, \boldsymbol{o_u}\}$. The $m \times n$ matrix R_u is called the *access* (*reference*) *matrix* [19] and the m-dimensional vector $\boldsymbol{o_u}$ is called the *offset vector.* Two references (possibly to different arrays) $\{R_u, \boldsymbol{o_u}\}$ and $\{R_{u'}, \boldsymbol{o_{u'}}\}$ are called *uniformly generated references* (UGR) [6] if $R_u = R_{u'}$.

Temporal reuse is said to occur when a reference in a loop nest accesses the same data in different iterations. Similarly, if a reference accesses nearby data, i.e., data residing in the same coherence unit (e.g., a cache line or a memory page), in different iterations, we say that *spatial reuse* occurs.

A data transformation can be defined as a transformation of array index space [11]. While, in principle, such a transformation can be quite general, in this work, we focus on the transformations that can be represented using linear transformation matrices. If $\mathcal{G}_u$ is the *array index space* (that is, the space that contains all possible array indices within array bounds) for a given m-dimensional array U, a data transformation causes an element $\boldsymbol{g} \in \mathcal{G}$ to be mapped to $\boldsymbol{g'} \in \mathcal{G}'$, where $\mathcal{G}'$ is the new (transformed) array index space. Such a data transformation can be represented by a pair $\{M_u, \boldsymbol{f_u}\}$, where M_u is an $m \times m'$ matrix and $\boldsymbol{f_u}$ is an m'-dimensional vector. The impact of such a data transformation is that a reference such as $\{R_u, \boldsymbol{o_u}\}$ is transformed to $\{R'_u, \boldsymbol{o'_u}\}$, where $R'_u = M_u R_u$ and $\boldsymbol{o'_u} = M_u\boldsymbol{o_u} + \boldsymbol{f_u}$. Most of the previous data transformation frameworks proposed in literature handle the special case where $m = m'$. The theory of determining suitable data transformation for a given array variable and post-transformation code generation techniques have been discussed in [11,8].

3 Array Unification

3.1 Approach

Consider the following loop nest that accesses two one-dimensional arrays using the same subscript function:

```
for $i$ = 1, N
  $b+ = U_1[i] + U_2[i]$
```

If considered individually, each of the references in this nest has perfect spatial locality as successive iterations of the i loop access consecutive elements from each array. However, if, for example, the base addresses of these arrays happen to cause a conflict in cache, the performance of this nest can degrade dramatically. In fact, in this code, it might even be possible to obtain a miss rate of 100%. The characteristic that leads to this degradation is that, between two successive accesses to array U_1, there is an intervening access from U_2, and vice versa. In other words, there is an iteration distance of 1 between successive uses of the same cache line, and during this duration, the said cache line *may* be evicted from cache due to the intervening access to the other array.

Let us now consider the following mapping of arrays U_1 and U_2 to the common data space (array) X.

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_2[i] \longrightarrow X[2i],$$

in which case we can re-write the loop nest as follows:[1]

```
for $i$ = 1, N
 $b+ = X[2i-1] + X[2i]$
```

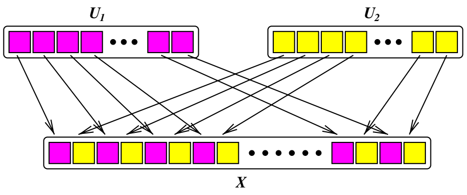

Fig. 1. Unification of two one-dimensional arrays.

A pictorial representation of these mappings is given in Figure 1. Note that in this new nest, for a given loop iteration, two references access two consecutive array elements; that is, the reuse distance between two successive accesses to the same cache line is potentially reduced to 0. Note that the same transformation can be done with multidimensional arrays as well. As an example, consider the following two-level nested loop:

```
for $i$ = 1, N
 for $j$ = 1, N
  $c+ = U_1[i][j] + U_2[i][j]$
```

Again, considering each reference separately, we have (potentially) perfect spatial locality (assuming that the arrays have a *row-major* memory layout).

[1] Note that, here, for the first transformation, we have $M_{u_1} = [2]$ and $\boldsymbol{f_{u_1}} = -1$ whereas for the second transformation, $M_{u_2} = [2]$ and $\boldsymbol{f_{u_2}} = 0$.

However, conflict misses may prevent this nest from attaining this locality at runtime. This would occur, for example, when two references have the same linearized stride and have the base addresses that map close together in cache. If we use the transformations

$$U_1[i][j] \longrightarrow X[i][2j-1] \text{ and } U_2[i][j] \longrightarrow X[i][2j],$$

we obtain the following nest:

```
for $i$ = 1, N
 for $j$ = 1, N
  $c+ = X[i][2j-1] + X[i][2j];$
```

Note that this transformation can be viewed as converting two N×N arrays to a single N×2N array. In general, if we want to transform k two-dimensional arrays U_i[N][N] ($1 \leq i \leq k$)— all accessed with the same subscript expression $[i][j]$ —to a single two-dimensional array X[N][kN], we can use the following generic data transformation:

$$M_{u_i} = \begin{bmatrix} 1 & 0 \\ 0 & k \end{bmatrix}, \ \boldsymbol{f_{u_i}} = \begin{bmatrix} 0 \\ i-k \end{bmatrix}.$$

Note that this transformation strategy can also be used whenreferences have subscript expressions of the form $[i \mp a][j \mp b]$. So far, we have implicitly assumed that the references accessed in the nest are uniformly generated. If two arrays U_1 and U_2 are accessed using references $\{R_{u_1}, \boldsymbol{o_{u_1}}\}$ and $\{R_{u_2}, \boldsymbol{o_{u_2}}\}$, respectively, where R_{u_1} and R_{u_1} are *not* necessarily the same, we need to select two data transformations $\{M_{u_1}, \boldsymbol{f_{u_1}}\}$ and $\{M_{u_2}, \boldsymbol{f_{u_2}}\}$ such that the transformed references $\{M_{u_1}R_{u_1}, M_{u_1}\boldsymbol{o_{u_1}} + \boldsymbol{f_{u_1}}\}$ and $\{M_{u_2}R_{u_2}, M_{u_2}\boldsymbol{o_{u_2}} + \boldsymbol{f_{u_2}}\}$ will access consecutive data items for a given iteration. Also, when considered individually, each reference should have good spatial locality. As an example, for the following nest, we can use the transformations

$$M_{u_1} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}, \ \boldsymbol{f_{u_1}} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \ ;$$

$$M_{u_2} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}, \ \boldsymbol{f_{u_2}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

```
for $i$ = 1, N
 for $j$ = 1, N
  $c+ = U_1[i][j] + U_2[j][i]$
```

After these transformations, the references inside the nest will be $X[i][2j-1]$ and $X[i][2j]$ instead of $U_1[i][j]$ and $U_2[j][i]$, respectively, exhibiting high group-spatial reuse. *It should be noted that such transformations actually involve both independent array transformations and array unification.* To see this, we can think of M_{u_2} as a composition of two different transformations. More specifically,

$$M_{u_2} = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

It should be noted that array unification can also be used to optimize locality performance of references with non-affine subscript functions. In fact, as noted by Leung and Zahorjan [11], data layout optimizations promise a wider applicality than loop-oriented techniques as far as non-affine references are concerned. This is because, unlike loop restructuring, the legality of data transformations do not rely on dependence analysis whereas the existence of even a single non-affine reference may prevent the candidate loop transformation which would otherwise improve data locality.

Let us consider the loop shown below assuming that $f(.)$ is a non-affine expression (which might even be an index array).

```
for $i$ = 1, N
 $b+ = U_1[f(i)] + U_2[f(i)]$
```

If considered individually, each reference here may have very poor locality as there is no guarantee that the function $f(.)$ will take successive values for successive iterations of i. In the worst case, this loop can experience 2N cache misses. Now, consider the following data transformations:

$$U_1[f(i)] \longrightarrow X[f(i),0] \text{ and } U_2[f(i)] \longrightarrow X[f(i),1],$$

assuming that the new array X has a row-major layout. It is easy to see that, due to the high probability that $X[f(i),1]$ and $X[f(i),2]$ will use the same cache line, we might be able to reduce the number of misses to N. Notice that the data transformations used here can be represented by

$$M_{u_1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \boldsymbol{f_{u_1}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } M_{u_2} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \boldsymbol{f_{u_2}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

It should be noted that this transformation can reduce capacity as well as conflict misses.

There is an important difference between such transformations and those considered earlier for the affine-references case. The data transformation matrices used for non-affine references need to be non-square. Of course, nothing prevents us to use non-square transformations for affine references as well. Consider the two-level nested loop shown above that has the references $U_1[i][j]$ and $U_2[i][j]$. Instead of using the transformations

$$U_1[i][j] \longrightarrow X[i][2j-1] \text{ and } U_2[i][j] \longrightarrow X[i][2j],$$

we could have easily used

$$U_1[i][j] \longrightarrow X[i][j][0] \text{ and } U_2[i][j] \longrightarrow X[i][j][1].$$

Therefore, in general, we have the options of using a two-dimensional array of size N×2N or a three-dimensional array of size N×N×2. The theory developed in the rest of the paper can easily be adapted to the latter case as well.

A group of arrays with a unification characteristic is called an *interest group.* The *unification characteristic* of the arrays in the same interest group is that they have the same dimensionality and each dimension has the same extent and the arrays are accessed with the *same frequency* in the innermost loop. The first two requirements are obvious. As for the third requirement, we say that two references are accessed with the same frequency if (in their subscript functions) they use exactly the same set of loop index variables. For instance, for a three-level nested loop with indices i, j, and k from top, the references $U_1[i+k][j]$ and $U_2[i][j-k]$ are accessed with the same frequency whereas the references $U_1[i][j]$ and $U_2[j][k]$ are *not* accessed with the same frequency. The problem with this last pair of references is that the reference $U_2[j][k]$ traverses the array at a much faster rate than the reference $U_1[i][j]$ (as the former has the innermost loop index k). It is known that a majority of the severe inter-variable conflict misses occur between the references that are accessed with the same frequency [17]. Later in the paper, we discuss more relaxed unification characteristics. Let us assume for now, without loss of generality, that we have only one interest group (the approach to be presented can be applied to each interest group independently), and, that each nested loop in a given code accesses a subset of the arrays in this interest group. Our objective is to determine a unification strategy so that the overall data locality of the code is improved. Our approach makes use of a graph structure where *nodes* represent *array variables* and *edges* represent the *transitions* between them. Let us first consider the example code fragment below to motivate our approach.

```
for $i$ = 1, N
  $b+ = U_1[i] + U_2[i]$
for $i$ = 1, N
  $c+ = U_3[i] + c$
```

If we transform only the arrays U_1 and U_2 using the mappings

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_2[i] \longrightarrow X[2i],$$

we will have the statement $b+ = X[2i-1] + X[2i]$ in the first nest. It is easy to see that, after these transformations, we have perfect spatial locality in both the nests. On the other hand, if we transform only U_1 and U_3 using the mappings

$$U_1[i] \longrightarrow X[2i-1] \text{ and } U_3[i] \longrightarrow X[2i],$$

we will have $b+ = X[2i-1] + U_2[i]$ in the first nest and $c+ = X[2i] + c$ in the second. The problem with this transformed code is that the references $X[2i-1]$ and $X[2i]$ iterate over the array using a step size of 2 and they are far apart from each other. It is very likely that a cache line brought by $X[2i-1]$ will be kicked off the cache by the reference $U_2[i]$ in the same or a close-by iteration. Finally, let us consider transforming all three arrays using

$$U_1[i] \longrightarrow X[3i-2], U_2[i] \longrightarrow X[3i-1], \text{ and } U_3[i] \longrightarrow X[3i].$$

In this case, we will have $b+ = X[3i-2] + X[3i-1]$ in the first nest and $c+ = X[3i] + c$ in the second nest. Note that in the second nest, we traverse the array using step size of 3 which may lead to poor locality for, say, a cache line size that can hold at most two elements. The preceding discussion shows that (1) selecting the subset of arrays to be unified (from an interest group) might be important, and that (2) it might not always be a good idea to include all the arrays (even in a given interest group) in the unification process. For instance, in the current example, the best strategy is to unify only the arrays U_1 and U_2. It is also easy to imagine that the use of the same array variable in different loop nests can make the problem even more difficult.

3.2 Representation

In order to capture the temporal relations between array variables globally, we use an undirected graph called *array transition graph* (ATG). In a given ATG(V, E), V represents the set of array variables used in the program, and there is an edge $e = (v_1, v_2) \in E$ with a weight of $w(e)$ if and only if there are $w(e)$ *transitions* between the arrays represented by v_1 and v_2. A transition between v_1 and v_2 corresponds to the case that the array variable represented by v_2 is touched immediately after the array variable represented by v_1 is touched, or vice versa. Although an ATG can be built by instrumenting the program and counting the number of transitions between different array variables, in this paper, we construct an ATG using the information (constant or symbolic) available at compile-time.

Consider the following code fragment, whose ATG is given in Figure 2(a).

```
for i = 1, N
  $U_5[i] = (U_3[i] * U_4[i]) + (U_1[i] * U_2[i])$
for i = 1, N
  $U_6[i] = (U_1[i] + U_3[i]) * U_5[i]$
for i = 1, N
  $U_4[i] = (U_5[i] - U_6[i]) * U_1[i]$
```

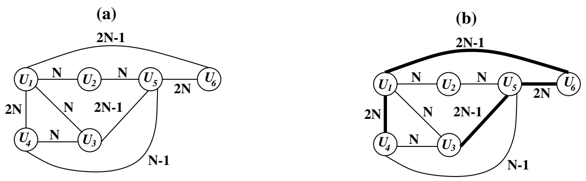

Fig. 2. (a) Address Transition Graph (ATG). (b) An important path.

As explained earlier, in a given ATG, the edge weights represent the number of transitions. For example, the weight of the edge between U_1 and U_4 is 2N, indicating that there are 2N transitions between these two array variables. N of these transitions come from the first nest where U_1 is accessed immediately after

U_4. The remaning transitions, on the other hand, come from the last nest where U_4 is written after U_1 is touched. Note that it makes no difference in which order we touch the variables as long as we touch them consecutively. An important thing to note about the ATG shown is that the edge weights differ from each other and it makes sense to unify the arrays with large edge weights as these are the arrays that are most frequently accessed one after another. For example, we have `2N` transitions between the variables U_1 and U_4 whereas we have only `N-1` transitions between U_4 and U_5, which indicates that it is more important to unify U_1 and U_4 than unifying U_4 and U_5. The problem then is to select a unification order (basically, paths in the ATG) such that as many high weight edges as possible should be covered. Note that this problem is similar to the state assignment with minimum Hamming distance problem, therefore, a polynomial-time solution is unlikely to exist. In fact, Liao shows that a similar problem (called *offset assignment* that involves selecting suitable placement order for scalar variables) is NP-complete [13]. Eisenbeis et al. [5] also use a similar graph-based representation. In the following, we present a polynomial-time heuristic which we found very effective in cases encountered in practice. Also note that in case a different memory access pattern is imposed by back-end, the code can be profiled first to extract this access pattern, and then, based on that, a suitable ATG can be constructed.

We start by observing that (after unification) a given array variable can have only *one left neighbor* and *one right neighbor.* If we take a closer look at the problem and its ATG, we can see that it is important to capture only *the paths with high weights.* After we detect a path that contains edges with high weights (henceforth called *important path*), then we can unify the array variables on this path. Note that a given ATG may lead to multiple important paths. Figure 2(b) shows an important path (using edges marked bold) that contains U_4, U_1, U_6, U_5, and U_3. Consequently, a solution for this example is to unify these variables in that order. The transformed program is as follows:

```
for $i$ = 1, N
  $X[5i-1] = (X[5i] * X[5i-4]) + (X[5i-3] * U_2[i])$
for $i$ = 1, N
  $X[5i-2] = (X[5i-3] + X[5i]) * X[5i-1]$
for $i$ = 1, N
  $X[5i-4] = (X[5i-1] - X[5i-2]) * X[5i-3]$
```

3.3 Formulation

We now present our strategy for determining the important paths whose vertices are to be unified. Given an ATG and a number of important paths on it, the cost of a unification strategy can be defined as the sum of the weights of the edges that do *not* belong to any important path. Let us first make the following formal definitions:

Definition 1 *Two paths are said to be 'disjoint' if they do not share any vertices.*

Definition 2 *A 'disjoint path cover' (which will be referred to as just a 'cover') of an $ATG(V, E)$ is a subgraph $C(V, E')$ such that, for every vertex v in C, we have degree(v) < 3, and there are no cycles in C (degree(.) denotes the number of edges incident on a vertex).*

Definition 3 *The 'weight' of a cover C is the sum of the weights of all the edges in C. The 'cost' of a cover C is the sum of the weights of all edges in G but not in C:*

$$cost(C) = \sum_{(e \in E) \wedge (e \notin C)} w(e)$$

A cover contains a number of important paths and the array variables in each important path can be unified. It can be seen that the cost of a unification is the number of adjacent accesses to array variables that are *not* adjacent in an important path. Given the definitions above, if a maximum weight cover for an ATG is found, then that also means that the minimum cost unification has also been found. The thick lines in Figure 2(b) show a disjoint path cover for the ATG given in Figure 2(a). The cost of this cover is `5N-1` which can be seen from the edges not in the cover.

Our unification problem can modeled as a graph theoretic optimization problem similar to Liao's [13] modeling of the offset assignment problem for DSP architectures and can be shown to be equivalent to the Maximum Weighted Path Cover (MWPC) problem. This problem is proven to be NP-hard. Next, a heuristic solution to the unification problem is given.

The heuristic is similar to Kruskal's spanning tree algorithm. Taking the ATG as input, it first builds a list of sorted edges in descending order of weight. The cover to be determined initially is empty. In each iteration, an edge with the maximum weight is selected such that the selected edges never form a cycle and no node will be connected to more than two selected edges. Note that the approach iterates at most $V - 1$ times and its complexity is $O(|E| \log |E| + |V|)$.

We now focus on different unification characteristics that are more relaxed than the one adopted so far, and discuss our approach to each of them. Specifically, we will consider three cases:

(1) Arrays that are accessed with the same frequency and have the same dimensionality and have the same dimension extents up to a permutation can be part of an important path (that is, they can be unified).
(2) Arrays that are accessed with the same frequency and have the same dimensionality but with different dimension extents can be part of an important path.
(3) Arrays that are accessed with the same frequency but have different dimensionalities can be part of an important path.

An example of the first case is the following two-level nested loop where arrays U_1 and U_2 are of sizes `N`$\times$`M` and `M`$\times$`N`, respectively.

```
for $i$ = 1, N
 for $j$ = 1, M
  $b += U_1[i][j] + U_2[j][i]$
```

Note that this first case is quite common in many floating-point codes. As before, we want to transform the references $U_1[i][j]$ and $U_2[j][i]$ to $X[i][2j-1]$ and $X[i][2j-1]$, respectively. Our approach to this problem is as follows. Since the dimension extents of the arrays U_1 and U_2 are the same up to a permutation, we first use a data transformation to one of the arrays (say U_2) so that the dimension extents become the same. Then, we can proceed with our usual transformation and map both the arrays to the common domain X.

In the second case, we allow the dimension extents of the arrays to be unified to be different from each other. Consider the example below assuming that the arrays U_1 and U_2 are of $\mathtt{N}\times\mathtt{N}$ and $\mathtt{M}\times\mathtt{M}$, respectively, and $\mathtt{N}\leq\mathtt{M}$.

```
for $i$ = 1, N
 for $j$ = 1, N
  $b += U_1[i][j] + U_2[j][i]$
```

In this case, we can consider at least two options. First, we can unify the arrays considering the largest extent in each dimension. In the current example, this corresponds to mappings

$$U_1[i][j] \longrightarrow X[i][2j-1] \text{ and } U_2[i'][j'] \longrightarrow X[i'][2j'],$$

for $1 \leq i, j \leq \mathtt{N}$ and $1 \leq i', j' \leq \mathtt{N}$. On the other hand, for $\mathtt{N}+1 \leq i', j' \leq \mathtt{M}$, we can set $X[i'][2j']$ to arbitrary values as these elements are never accessed. A potential negative side effect of this option is that if the elements $X[i][2j-1]$ (when $\mathtt{N}+1 \leq i, j \leq \mathtt{M}$) happen to be accessed in a separate loop, the corresponding cache lines will be underutilized. The second option is to transform only the portions of the arrays that are used together. For instance, in our example, we can divide the second array above (U_2) into the following three regions:

$$U_{21}[i][j](1 \leq i, j \leq \mathtt{N}), U_{22}[i][j](\mathtt{N}+1 \leq i \leq \mathtt{M}, 1 \leq j \leq \mathtt{N}),$$

$$\text{and } U_{23}[i][j], (1 \leq i \leq \mathtt{N}, \mathtt{N}+1 \leq j \leq \mathtt{M}).$$

After this division, we can unify U_1 with only U_{21} as they represent the elements that are accessed concurrently.

To handle the third case mentioned above, we can use array replication. We believe that the techniques developed in the context of cache interference minimization (e.g., [17]) can be exploited here. The details are omitted due to lack of space.

4 Experiments

The experiments described in this section were performed on a single R10K processor (195 MHz) of the SGI/Cray Origin 2000 multiprocessor machine. The processor utilizes a two-level data cache hierarchy: there is a 32 KB primary (L1) cache and a 4 MB secondary (L2) cache (unified). The access time for the L1 cache is 2 or 3 clock cycles and that for the L2 cache is 8 to 10 clock cycles. Main memory access time, on the other hand, ranges from 300 to 1100 nanoseconds (that is 60 to 200+ cycles).

The proposed framework has been implemented within the Parafrase-2 compiler. The current implementation has the following limitations. First, it uses a restricted form of unification characteristic. It does not unify two arrays unless they are accessed with the same frequency and are of the same dimensionality and have the same extents in each dimension. Second, it works on a single procedure at a time. It does not propagate the memory layouts across procedures, instead it simply transforms the unified arrays explicitly between procedure boundaries at runtime. Finally, except for procedure boundaries, it does not perform any dynamic layout transformation. In other words, an array is never unified with different groups of arrays in different parts of the code.

We use two benchmark codes: `bmcm` and `charmm`. The `bmcm` is a *regular* array-dominated code from the Perfect Club benchmarks. We report performance numbers for six different versions of the code: `noopt`, `dopt`, `unf`, `noopt+`, `dopt+`, and `unf+`. Each version is eventually compiled using the native compiler on the Origin. `noopt` is the unoptimized (original) code, and `dopt` is the version that uses individual data transformations for each array. It is roughly equivalent to data-centric optimization schemes proposed in [8] and [2]. The version `unf`, on the other hand, corresponds to the code obtained through the technique explained in this paper. These three versions do not use the native compiler's locality optimizations, but use back-end (low-level) optimizations. The versions `noopt+`, `dopt+`, and `unf+` are the same as `noopt`, `dopt`, and `unf`, respectively, except that the entire suite of locality optimizations of the native compiler is activated.

Figure 3 shows the execution times (seconds), MFLOPS rates, L1 hit rates, and L2 hit rates for these six versions using two different input sizes (small=$\sim$6 MB and large=$\sim$24 MB). We see that for both small and large input sizes, `unf` performs better than `noopt` and `dopt`, indicating that, for this benchmark, array unification is more successful than transforming memory layouts independently. In particular, with a 14 MB input size, `unf` reduced the execution time by around 67% over the `dopt` version. We also note that the technique proposed in this paper is not sufficient alone to obtain the best performance. For instance, with the large input size, `noopt+` generates 131.20 MFLOPS which is much better than 94.78 MFLOPS of `unf`. However, applying tiling brings our technique's performance to 186.77 MFLOPS. These results demonstrate that control-centric transformations such as tiling are still important even in the existence of array unification. Finally, we also note that the `dopt` version takes very little advantage of the native compiler's powerful loop optimizations.

We next focus on a kernel computation from an irregular application, `charmm`. This code models the molecular dynamic simulation; it simulates dynamic interactions (bonded and nonbonded) among all atoms for a specific period of time. As in the previous example, we experiment with two different input sizes

	∼6 MB						∼24 MB					
	noopt	noopt+	dopt	dopt+	unf	unf+	noopt	noopt+	dopt	dopt+	unf	unf+
Execution Time (sec):	2.169	0.793	1.890	0.791	1.409	0.641	49.964	7.631	32.734	6.928	10.627	5.443
MFLOPS:	56.45	176.52	63.17	176.28	85.62	215.33	20.07	131.20	30.59	145.27	94.78	186.77
L1 Hit Rate (%):	82.8	94.8	87.5	93.5	99.8	99.4	55.9	95.2	87.2	92.4	89.8	99.4
L2 Hit Rate (%):	96.9	98.7	99.6	99.3	98.6	99.6	95.2	98.0	82.0	99.3	95.8	97.9

Fig. 3. Performance results for `bmcm`.

(small=∼240 KB and large=∼12 MB). Since `dopt` version of this code is same as the `noopt` version, we omit the former from consideration. We can make two important observations from Figure 4. First, as expected, control-centric locality optimizations of the native compiler bring only a marginal benefit, most of which is due to scalar replacement. Second, the L1 miss rates for the `noopt` version are quite high due to irregular nature of the data access pattern exhibited by the kernel. A simple array unification which maps two most frequently accessed (through indexed arrays) arrays into a single array improves both miss rates and performance.

Overall, for these two codes, we find array unification to be very successful.

	∼240 KB				∼12 MB			
	noopt	noopt+	unf	unf+	noopt	noopt+	unf	unf+
Execution Time (sec):	0.964	0.961	0.804	0.809	240.445	240.429	188.941	187.055
MFLOPS:	127.93	128.07	146.43	145.793	119.98	119.93	167.82	170.11
L1 Hit Rate (%):	70.6	75.5	90.1	90.0	85.2	90.5	95.5	95.3
L2 Hit Rate (%):	98.1	96.0	98.3	98.0	83.1	91.8	96.3	96.8

Fig. 4. Performance results for a kernel code from `charmm`.

5 Related Work

Wolf and Lam [19] present a locality optimization framework that makes use of both unimodular loop transformations as well as tiling. Li [12] proposes a locality optimization technique that models the reuse behavior between different references to the same array accurately. McKinley, Carr, and Tseng [14] present a simpler but very effective framework based on a cost (cache miss) model. Several other researchers propose different tiling algorithms with methods to select suitable tile sizes (blocking factors) [7,10]. The effectiveness of these techniques is limited by the inherent data dependences in the code.

More recently, a number of researchers have addressed limitations of loop-based transformations, and proposed data layout optimizations called data transformations. Leung and Zahorjan [11] and Kandemir et al. [8] propose data transformations that apply individual layout optimizations for each array. While such transformations are known to reduce some portion of conflict misses (in addition to capacity misses) due to improved spatial locality in the inner loop positions, they are not very effective to reduce inter-variable conflict misses. To reduce the severe impact of conflict misses, Rivera and Tseng propose array padding [18].

Many research groups have also focused on combining loop and data transformations in an integrated framework. Cierniak and Li [2], Kandemir et al. [9], and O'Boyle and Knijnenburg [16] investigate the integrated use of loop and data transformations to enhance data locality. The large search space for such integrated approaches generally forces them to limit the number of potential transformations. The data-centric array unification framework used in this paper can also be integrated with loop transformations (although this issue is beyond the scope of this paper).

Recently, a number of runtime-based approaches have been proposed to improve locality of irregular array applications. Mellor-Crummey et al. [15], and Ding and Kennedy [3] present data re-ordering techniques for irregular applications. While these techniques specifically focus on irregular applications, the framework proposed in this paper can be used for both irregular and regular applications.

Ding and Kennedy [4] present an inter-array data regrouping strategy to enhance locality. The main idea is to pack useful data into cache lines so that all data elements in a cache line are consumed before the line is evicted from the cache. Our work is different from theirs in several aspects. First, their implementation does not group arrays unless they are always accessed together. In contrast, our approach has a more *global view*, and considers the entire procedure with different nests accessing different subsets of the arrays declared. Second, since we formulate the problem within a mathematical framework, we can easily integrate array unification with existing loop and data transformations. Third, it is not clear in their work how the arrays with multiple and different (nonuniformly generated) references are handled.

6 Conclusions and Future Work

We have described a data optimization scheme called array unification. Our approach is based on a mathematical framework that involves arrays, access patterns, and temporal access relations between different array variables. The general problem is formulated on a graph structure and temporal access relations are captured by this representation. Subsequently, a heuristic algorithm determines the array variables to be unified and maps them into a common array (data) space.

This work can be extended in a number of ways. First, we need a solid mechanism to integrate array unification with loop-based transformation techniques. Second, the current approach works only on a single procedure at a time. A mechanism that propagates the new array spaces (after unification) across procedure boundaries would further improve the performance.

References

1. T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In Proc. *the SIGPLAN'99 Conf. on Prog. Lang. Design and Impl.,* Atlanta, GA, May 1999.

2. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. SIGPLAN '95 Conf. on Programming Language Design and Implementation*, June 1995.
3. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at runtime. In Proc. *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation,* Georgia, May, 1999.
4. C. Ding and K. Kennedy. Inter-array data regrouping. In Proc. *the 12th Workshop on Languages and Compilers for Parallel Computing,* San Diego, CA, August 1999.
5. C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In Proc. *the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.
6. D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel & Distributed Computing*, 5(5):587–616, October 1988.
7. F. Irigoin and R. Triolet. Super-node partitioning. In *Proc. 15th Annual ACM Symp. Principles of Prog. Lang.*, pp. 319–329, San Diego, CA, January 1988.
8. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane based approach for optimizing spatial locality in loop nests. In Proc. *1998 ACM Intl. Conf. on Supercomputing,* Melbourne, Australia, July 1998.
9. M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM Intl. Conf. on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.
10. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.
11. S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. *Technical Report TR 95-09-01,* Dept. Computer Science and Engineering, University of Washington, Sept. 1995.
12. W. Li. *Compiling for NUMA parallel machines*. Ph.D. Thesis, Cornell Uni., 1993.
13. S. Y. Liao. *Code Generation and Optimization for Embedded Digital Signal Processors.* Ph.D. Thesis, Dept. of EECS, MIT, Cambridge, Massachusetts, June 1996.
14. K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996.
15. J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications. In Proc. *the ACM Intl. Conf. on Supercomputing,* Rhodes, Greece, June 1999.
16. M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, October 1998, Paris, France.
17. O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In Proc. *the IEEE Supercomputing'93,* Portland, November 1993.
18. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, Montreal, Canada, June 1998.
19. M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Prog. Lang. Design and Implementation*, pages 30–44, June 1991.