ARRAY COMPOSITION AND DECOMPOSITION FOR OPTIMIZING EMBEDDED APPLICATIONS†

G. Chen, M. Kandemir, A. Nadgir

Pennsylvania State University
University Park, PA 16802, USA
{guilchen, kandemir, nadgir}@cse.psu.edu

U. Sezer

University of Wisconsin
Madison, WI 53706, USA
sezer@ece.wisc.edu

ABSTRACT

Optimizing array accesses is extremely critical in embedded computing as many embedded applications make use of arrays (in form of images, video frames, etc). Previous research considered both loop and data transformations for improving array accesses. However, data transformations considered were mostly limited to linear data transformations and array interleaving. In this paper, we introduce two data transformations: array decomposition (breaking up a large array into multiple smaller arrays) and array composition (combining multiple small arrays into a single large array). This paper discusses that it is feasible to implement these optimizations within an optimizing compiler.

1. INTRODUCTION

Two widely used ways of optimizing array-based embedded applications are loop and data transformations. Loop transformations change the execution order of loop iterations for achieving a better data access pattern from the data locality and/or parallelism viewpoints [1, 6, 7, 9]. While these transformations are widely used in optimizing compilers from industry and academia, they have several important drawbacks. First, a loop transformation is restricted by data dependences; that is, one may not change the execution order of loop iterations in an arbitrary fashion. Second, a loop transformation affects all the array references enclosed by the loop; while some of these references have better access patterns after the transformation, access patterns of others may actually get worse. Third, some complex loop structures (e.g., while-loops and imperfectly-nested loops) may prevent the best loop transformation from being applied, leading to suboptimal results.

These drawbacks motivated compiler researchers to focus on alternative optimization strategies. For example, data transformations modify the order in which array elements are stored in memory [1, 2, 5, 8]. Targeting memory layouts instead of loop iterations allows us to disregard data dependences between loop iterations. In addition, a data transformation can easily be applied to imperfectly-nested loop structures and while-loops. And, finally, each array can be layout-transformed independently from the others. These advantages enable data transformations to generate better results than loop transformations in some circumstances. One major difficulty in applying data transformations is that we need to consider all references to the array to be transformed in all loop nests.

Most of the previous work on data transformations focused on two types of transformations: linear data transformations [5, 8] and array interleaving [3]. In linear data transformations, each array is layout-transformed in isolation. An example would be converting a row-major memory layout to a column-major one (if doing so is beneficial from the data locality viewpoint). While this approach might be successful in some cases, in certain programs it can fail to capture the interaction between different arrays. Array interleaving, in comparison, interleaves two or more arrays accessed using the same (or similar) strides, usually to eliminate inter-array conflict misses. However, there are still several cases for which one might want to employ a more general data transformation:

- *Breaking a large array into multiple smaller arrays.* Such a transformation can be very useful to enhance data cache behavior. This is because working with a large array tends to pollute cache in a short period of time. For example, suppose that we want to access a small rectangular portion of a large two-dimensional array. In a row-major or a column-major memory layout, trying to access this portion will also bring several irrelevant array elements into the data cache (as cache transfers occur at a block granularity). On the other hand, if we can somehow store the elements in the array portion we are interested in consecutively in memory, we can achieve a much better cache behavior (as we do not bring unwanted elements to the cache).
- *Combining multiple arrays into one large array.* This is the opposite of the transformation summarized in the previous bullet. Combining arrays is a more general transformation than interleaving them as we can combine arrays in many different (and perhaps even asymmetric) ways. For example, we can simply attach one array next to another, or embed one array within another. Achieving this using just array interleaving and linear loop transformations alone is not easy (if at all possible).
- *Changing array structure of an application.* Sometimes, one might want to change the underlying array structure of the program without modifying the access pattern. For example, an application can have four three-dimensional arrays, and we may want to re-code the application using one two-dimensional array and two four-dimensional arrays (keeping the total number of array elements constant). We can achieve this by first decomposing the arrays (the first bullet) and then re-composing the resulting small arrays (the second bullet). Note that being able to transform shapes of data arrays in a user-transparent manner (e.g., through a compiler) increases the productivity of the programmer.

It should be noticed that these are just three example scenarios where array layout (data) transformations more general than linear transformations and array interleaving might be of use in practice. These transformations are based on array decomposition (i.e., breaking an array into multiple smaller arrays) and array composition (i.e., combining multiple arrays into one big array)

†This work was supported in part by NSF CAREER Award #0093082.

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

and are the focus of this paper. In order to apply array decomposition and/or composition, an optimizing compiler needs to make important decisions. The first is a policy decision and tries to answer the question of what data transformations to apply. While this is a very important question to address, it is not the main focus of this work. Instead, this paper mainly deals with the second decision, which addresses how the data transformations (defined by the policy) can be applied. This is the mechanism aspect of the overall problem, and this paper proposes a solution strategy based on polyhedral algebra. Specifically, we present a compiler-based strategy that changes the underlying array structure of a program under a given set of transformations.

2. PROBLEM DEFINITION

In this paper, we consider the case of references to arrays with affine subscript functions in nested loops, which are common in array-based embedded media applications [1]. Consider such an access (an array reference) to an m-dimensional array in an n-deep loop nest. Let $\vec{I}$ denote the iteration vector (consisting of loop indices starting from the outermost loop). Then, our array reference can be represented as $\mathcal{P}\vec{I} + \vec{o}$, where the $m \times n$ matrix $\mathcal{P}$ is called the access (or reference) matrix [9] and the m-element vector $\vec{o}$ is called the offset vector. Expression $\vec{J} = \mathcal{P}\vec{I} + \vec{o}$ is called the array index expression. As an example, for an array reference such as $A_1[i+1][i+j-2]$, we have

$$\vec{J} = \begin{pmatrix} i+1 \\ i+j-2 \end{pmatrix}.$$

The set of values that can be taken on by $\vec{I}$ defines the iteration space of the nest, and is denoted using $\mathcal{I}$.

In order to enumerate iteration sets and array elements, in this work, we employ a polyhedral tool called the Omega Library [4]. This library is particularly useful in our context as it can describe iteration spaces, data spaces, and mappings (relations) between them, and such relations can help us formalize and implement our data transformations.

A data mapping transforms one array into another, and this can be implemented using linear matrix transformations. Consequently, we can define the problem addressed in this paper as follows:

> Given an input application code and a set of array mappings $\mathcal{F} = \{\vec{f_1}, \vec{f_2}, ..., \vec{f_s}\}$, where each $\vec{f_j}$ is of the form
>
> $$\vec{f_j}: \quad A_j(\vec{J}) \longrightarrow A_k(\vec{K}),$$
>
> where $\vec{J_L} \leq \vec{J} \leq \vec{J_U}$, $\vec{K_L} \leq \vec{K} \leq \vec{K_U}$, and $\vec{K} = \mathcal{M}\vec{J} + \vec{\delta}$, transform (re-write) the code applying all array mappings in the set $\mathcal{F}$.

Here, $\vec{J_L}$ and $\vec{J_U}$ are the bounds of the region of interest for array A_j, and $\vec{K_L}$ and $\vec{K_U}$ are the corresponding bounds for array A_k. Also, the linear matrix $\mathcal{M}$ is called the data transformation matrix, and $\vec{\delta}$ is called the replacement vector.

Our objective is to re-write the code after the mappings given in $\mathcal{F}$ are applied. In re-writing the code, there are two important issues that need to be addressed. The first of these is re-writing the array declarations and the second one is re-writing the body of the code itself (i.e., re-writing the array references). Re-writing the array declarations can be directly performed using the mappings in $\mathcal{F}$, so, we mainly focus on re-writing the code body. We assume that the mappings given are reasonable. For instance, an original array element is mapped to only a single (new) array element. Although beyond the scope of this paper, Omega Library can be used to check this and similar constraints as well.

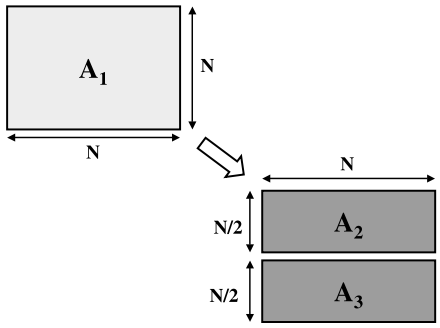

Figure 1: An example data transformation. In this example, we want to divide an $N \times N$ array A_1 into two smaller arrays (A_2 and A_3) such that each half of A_1 is mapped to a different array.

As an example, consider the code fragment below:

```
int  sum, A1[N][N];

for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    sum = sum + A1[i][j];
```

Assume that $\mathcal{F} = \{f_1, f_2\}$, where

$$f_1: \quad A_1[i][j] \longrightarrow A_2[i'][j']$$
$$f_2: \quad A_1[i''][j''] \longrightarrow A_3[i'''][j''']$$

and $0 \leq i < N/2$; $0 \leq j < N$; $0 \leq i' < N/2$; $0 \leq j' < N$; $N/2 \leq i'' < N$; $0 \leq j'' < N$; $0 \leq i''' < N/2$; and $0 \leq j''' < N$. That is, we would like to divide a given array A_1 into two smaller arrays (A_2 and A_3) such that each half of A_1 is mapped to a different array (see Figure 1).

Note that, in this case, the data transformation matrices can be written as:

$$\mathcal{M}_{12} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \mathcal{M}_{13} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Similarly, the replacement vectors are

$$\delta_{12} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ and } \delta_{13} = \begin{pmatrix} -N/2 \\ 0 \end{pmatrix}.$$

Consequently, the transformed code fragment can be written as follows:

```
int  sum, A2[N/2][N], A3[N/2][N];

for(i = 0; i < N/2; i++)
```

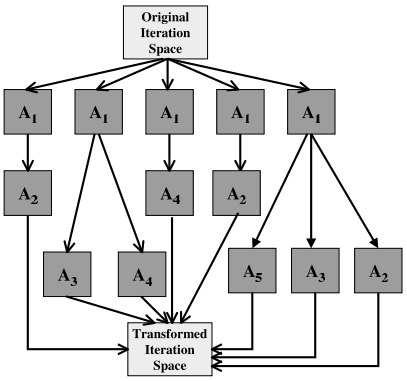

Figure 2: An example application of array decomposition along with the associated loop transformation. Note that loop transformation is necessary to preserve program correctness.

$$
\begin{aligned}
for(j = 0; j < N; j++)\\
sum = sum + A_2[i][j];\\
for(i = N/2; i < N; i++)\\
for(j = 0; j < N; j++)\\
sum = sum + A_3[i][j];
\end{aligned}
$$

3. OUR SOLUTION TO ARRAY DECOMPOSITION

In this section, we propose a solution strategy for re-writing the code after array decomposition. It is important to emphasize that the problem we are attacking is not trivial at all. Suppose, for example, that an array A_1 is used in a given loop nest and we have multiple references to it. Assuming that this array is to be broken up into multiple arrays, we need to determine, for each reference, which array to use. The problem is actually harder than this because, depending on the loop bounds and array subscript functions, a given (original) reference to A_1 may be mapped multiple references, each to a different array in the transformed code (e.g., see the example in the previous section where a single reference to A_1 has been transformed to two references: one to A_2 and one to A_3). To accommodate this, we need to restructure the loop nest in question. In other words, our approach needs to employ loop transformations as well.

Figure 2 illustrates the idea behind our approach to array decomposition using an example. In this example, in the original loop nest (iteration space), array A_1 is accessed (via five different references). We perform our optimization in two steps: In the first step, we replace the original array references in the code with references to the new arrays (A_2 through A_5). Note that it is possible that an original array reference can generate multiple references (as in the cases of the second and fifth references in the original code). We will explain shortly how to determine whether this is the case. It is also possible that different references in the original code (to the same array) can result in references to the same array in the transformed code. In the second step, we transform the iteration space of the loop (using loop transformations) to accommodate new array references. However, the loop transformation used here is loop splitting [10], and is always legal. It should be noted that the scenario given in this example involves only a single array. If there are multiple arrays in the original nest, then the first step discussed is repeated for each of them. However, in applying the second step, the interaction between arrays should be accounted for, since the loop iteration space is common to all arrays within it.

To implement the first step, we use the mapping functions given in set $\mathcal{F}$. Specifically, given a set of mappings $\{\vec{f_1}, \vec{f_2}, ..., \vec{f_s}\}$, and an array reference $A_j(\vec{J})$ in the original code that is accessed by loop iteration vector $\vec{I}$ (that is, subscript expression $\vec{J}$ is a function of $\vec{I}$), we first determine a set of mappings $\mathcal{F}' \in \mathcal{F}$ such that all the array elements accessed by $A_j(\vec{J})$ under $\vec{I}$ are included in $\mathcal{F}'$, and that $\mathcal{F}'$ is the smallest of such sets. In other words, adding one more mapping to $\mathcal{F}'$ will not change a thing. This $\mathcal{F}'$ set is referred to as the cover set for reference $A_j(\vec{J})$ under $\vec{I}$. The concept of cover set is very important, because if an array A_k is in the cover set (in the right-hand-side of a mapping in $\mathcal{F}'$), this means that it should be used in generating the new (transformed) code.

In formal terms, an array A_k is in the cover set of $A_j(\vec{J})$ under $\vec{I}$ iff

$$
\begin{aligned}
& \exists\, (\vec{I} \in \mathcal{I} \text{ and } a \in A_j \text{ and } f_k \in \mathcal{F}') \text{ such that } a = A_j(\vec{J})\\
\text{and} \quad & \vec{K} = \mathcal{M}\vec{J} + \vec{\delta} \text{ and } \vec{J} = \mathcal{P}\vec{I} + \vec{o} \text{ and } \vec{I} \in \mathcal{I}\\
\text{and} \quad & \vec{f_k}: \quad A_j(\vec{J}) \longrightarrow A_k(\vec{K}).
\end{aligned}
$$

In this expression, $\vec{J}$ is the subscript expression for A_j and $\mathcal{M}$ and $\vec{\delta}$ are the data transformation matrix and the replacement vector, respectively, for $\vec{f_k}$. This condition given above is called the coverage condition and it can easily be checked using Omega Library [4], which is the tool used in our implementation.

All arrays A_k that satisfy the condition above are in the cover set and are used to replace the original reference after the transformation. However, in order to know which part of the original region covered by $A_j(\vec{J})$ now needs to be covered by an A_k, we need to consider the bounds for f_k. More specifically, in $\vec{f_k}: \quad A_j(\vec{J}) \longrightarrow A_k(\vec{K})$, if for a given $\vec{I} \in \mathcal{I}$, $\vec{J} = \mathcal{P}\vec{I} + \vec{o}$ holds, where $\vec{J_L} \leq \vec{J} \leq \vec{J_U}$, $\vec{K_L} \leq \vec{K} \leq \vec{K_U}$, and $\vec{K} = \mathcal{M}\vec{J} + \vec{\delta}$, then for iteration $\vec{I}$ we should use A_k when transforming A_j. Note that this last condition can be checked using Omega Library.

The set of iterations $\vec{I_k} \in \vec{I}$ in which the reference $A_j(\vec{J})$ should be replaced (in the transformed code) by reference $A_k(\vec{K})$ is called the local iteration set (LIS) of A_k for A_j, and is denoted by $\mathcal{I}_{kj}$. Again, the elements of this set can be enumerated by Omega Library. Once the local iteration sets are determined, the loop in question is divided into LISs (using loop splitting), and for each LIS a different array (from $\mathcal{F}'$) is used. It should be observed, however, that the problem becomes more complex when there are multiple arrays in the (original) nest being optimized. This is because each original array in the code can demand a different loop splitting, and we need to reconcile these different demands somehow. The solution that we propose in this paper is based on determining the minimum-sized LISs. The idea can be best explained using an example. Suppose that a given nest accesses two arrays A_1 and B_1 (using a single reference per array) and that these arrays demand different splittings of the iteration space (loop splittings) as illustrated in Figure 3(a). Note that the first splitting has two LISs, whereas the second one has three LISs. In this case, our approach superimposes these two individual splittings and divides the (original) iteration space into four LISs (i.e., it generates the maximum number of LISs — also called the minimum-sized LISs), which is also shown in Figure 3(a). Now, assuming that the

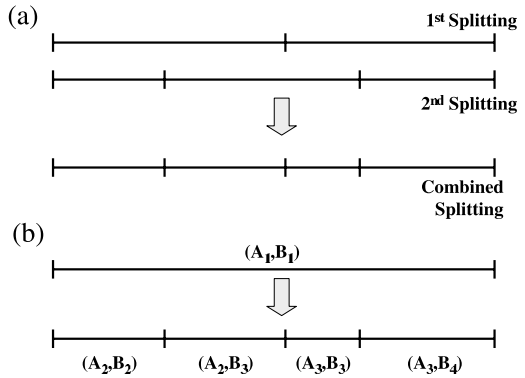

Figure 3: (a) Two different individual splittings and the combined splitting. Our approach superimposes these two individual splittings and divides the (original) iteration space into four LISs (i.e., it generates the maximum number of LISs) (b) Transforming array (reference)s.

two references that will replace A_1 are to A_2 and A_3 and that the three references that will replace B_1 are to B_2, B_3, and B_4, the four LISs in the resulting code have the following pairs of references: (A_2,B_2), (A_2,B_3), (A_3,B_3), and (A_3,B_4) as shown in Figure 3(b). Note that the original reference pair in this example was (A_1,B_1). One can notice that a drawback of this strategy is that it leads an increase in the code space (which might be problematic in an embedded environment). However, in practice, we do not expect a significant code size increase since most useful mappings induce only a few partitionings, and in many nests, there are references to only a few different arrays.

4. OUR SOLUTION TO ARRAY COMPOSITION

Array composition is opposite of array decomposition and combines multiple arrays into a single array. It should be noted, however, that one can have multiple compositions for the same code; that is, after the transformation, we may have more than one array (i.e., the transformed code may have multiple arrays). The polyhedral support required by array composition is very similar to the one that has been discussed above (Section 3) for array decomposition. Therefore, rather than repeating the entire framework here, we just highlight the important points. In fact, since in array composition (unlike array decomposition) an original reference can be transformed to only a single reference, the transformation process is simpler.

Suppose that we have the following two mappings to an array A_k:

$$\vec{f_j}: \quad A_j(\vec{J}) \longrightarrow A_k(\vec{K})$$
$$\vec{f_l}: \quad A_l(\vec{l}) \longrightarrow A_k(\vec{K'}).$$

where $\vec{J}_L \leq \vec{J} \leq \vec{J}_U$, $\vec{L}_L \leq \vec{L} \leq \vec{L}_U$, $\vec{K}_L \leq \vec{K} \leq \vec{K}_U$, $\vec{K'}_L \leq \vec{K'} \leq \vec{K'}_U$. If $\vec{K} = \mathcal{M}\vec{J} + \vec{\delta}$ and $\vec{K'} = \mathcal{M'}\vec{J} + \vec{\delta'}$, then we can determine the new subscript expressions as follows. For a given iteration point $\vec{I}$, we first determine which elements of A_j and A_l are accessed. After that, we apply that element the data transformations $(\mathcal{M}, \vec{\delta})$ and $(\mathcal{M'}, \vec{\delta'})$. Then, we check whether the resulting element falls in the range $[\vec{K}_L : \vec{K}_U]$ or in the range $[\vec{K'}_L : \vec{K'}_U]$. If it is the first range, we transform the original element by $(\mathcal{M}, \vec{\delta})$; otherwise, we employ $(\mathcal{M'}, \vec{\delta'})$. The code generation issues for array composition case are very similar to those for array decomposition case, and are not detailed here due to lack of space.

5. CONCLUDING REMARKS

The primary objective of a compiler that targets array/loop nest based applications is to transform iteration and data spaces for enhancing data locality and/or parallelism. In this study, we presented two data transformation techniques (array composition and array decomposition), which are more general than previous data transformation techniques. Our future work will focus on two different but related directions. First, we will develop several composition and decomposition algorithms and compare them to each other. Second, we will investigate ways of integrating our transformations with the large body of existing loop/data transformations.

6. REFERENCES

[1] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.

[2] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[3] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proc. the Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 4–6, 1999.

[4] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. *Technical Report CS–TR–3445*, CS Dept., University of Maryland, College Park, March 1995.

[5] S.-T. Leung and J. Zahorjan. Optimizing data locality by array restructuring. *Technical Report TR 95–09–01*, Dept. of Computer Science and Engineering, University of Washington, September 1995.

[6] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, NY, 1993.

[7] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[8] M. O'Boyle and P. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 287–297, Aachen, Germany, 1996.

[9] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[10] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, 1996.