Generalized Data Transformations for Enhancing Cache Behavior*

V. De La Luz, M. Kandemir, I. Kadayif
CSE Department
Penn State University
University Park, PA, 16802

U. Sezer
ECE Department
University of Wisconsin
Madison, WI 53706

Abstract

The performance gap between processors and off-chip memories has widened in the last years and is expected to widen even more. Today, it is widely accepted that caches improve significantly the performance of programs, since most of the programs exhibit temporal and/or spatial locality in their memory reference patterns. However, conflict misses can be a major obstacle preventing an application from utilizing the data cache effectively. While array padding can reduce conflict misses it can also increase the data space requirements significantly. In this paper, we present a compiler-based data transformation strategy, called the "generalized data transformations," for reducing inter-array conflict misses in embedded applications. We present the theory behind the generalized data transformations and discuss how they can be integrated with compiler-based loop transformations. Our experimental results demonstrate that the generalized data transformations are very effective in improving data cache behavior of embedded applications.

1. Introduction

In many array-intensive embedded applications, conflict misses can constitute a significant portion of total data cache misses. To illustrate this, we give in Figure 1, for an 8KB direct-mapped data cache, the breakdown of cache misses into conflict misses and other (capacity plus cold) misses for seven embedded applications from image and video processing.[1] We see that, on the average, conflict misses consist of 42.2% of total cache misses. In fact, in two applications (`Vcap` and `Face`), conflict misses constitute more than 50% of all misses. The reason for this behavior is the repetitive characteristic of conflict misses; that is, they tend to repeat themselves at regular (typically short) intervals as we execute loop iterations. The small associativities of the data caches employed in embedded systems also contribute to large number of conflict misses.

A well-known technique for reducing conflict misses is array padding [11]. This technique reduces conflict misses by affecting the memory addresses of the arrays declared in the application. It has two major forms: intra-array padding and inter-array padding. In intra-array padding, the array space is augmented with a few extra columns and/or rows to prevent the different columns/rows of the array from conflicting with each other in the data cache. For example, an array declaration such as $A(N, M)$ is modified to $A(N, M + k)$ where k is a small constant. This can help to prevent conflicts between two elements on different rows. Inter-array padding, on the other hand, inserts dummy array declarations between two original consecutive array declarations to prevent the potential conflict misses between these two arrays. For instance, a declaration sequence such as $A(N, M)$, $B(N, M)$ is transformed to $A(N, M)$, $D(k)$, $B(N, M)$, where D is a dummy array with k elements. This affects the array base addresses and can be used for reducing inter-array conflicts. While array-padding has been shown to be effective in reducing conflict misses in scientific applications [11], there is an important factor that needs to be accounted for when applying it in embedded environments: *increase in data space size.* Since data space demand of applications is the main factor that determines the capacity and cost of data memory configuration in embedded designs, an increase in data space may not be tolerable. To evaluate the impact of array padding quantitatively, we performed a set of experiments with our benchmark codes. Figure 2 gives the percentage reduction in (simulated) execution time (for an embedded MIPS processor with an 8KB data cache) and the percentage increase in data space when array padding is applied to our benchmarks. The specific array padding algorithm used to obtain these results is similar to that proposed by Rivera and Tseng [11] and represents the state-of-the-art. The values given in this graph are with respect to the original versions of codes. One can observe from these results that while array padding reduces the execution time by 10.3% on the average, it also increases the data space requirements significantly (15.1% on the average). These results motivate us for considering other techniques for eliminating conflict misses in embedded applications.

To see whether most of conflict misses come from intra-array conflicts (i.e., the different portions of the same array are conflicting in the data cache) or from inter-array conflicts (i.e., the portions of different arrays are conflicting), we also measured the contribution of each type of conflict misses. The results presented in Figure 3 indicate that the overwhelming majority of conflict misses occur between the different arrays (92.1% on the average). We believe that this is due to two main reasons. First, in many loop bodies in our benchmarks, there are only a small number of references to each array. Second, in many cases, the lengths of rows or columns are not large enough to create intra-array conflicts in the cache. Based on the results given in Figures 1, 2, and 3,

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

[1]The important characteristics of our benchmarks as well as detailed experimental results will be given later in the paper.

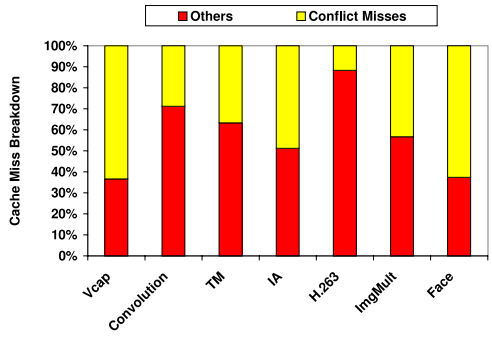

Figure 1. Contribution of conflict misses and other (capacity plus cold) misses.

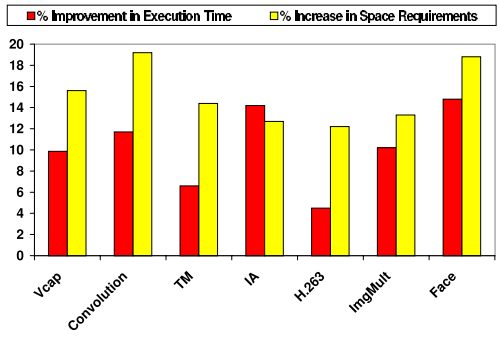

Figure 2. Impact of array padding.

we can conclude that a technique that focuses on minimizing inter-array conflict misses without increasing the data space requirements might be very desirable in embedded environments.

In this paper, we present such a *compiler-based* data transformation strategy for reducing inter-array conflict misses in embedded applications. This strategy maps the simultaneously used arrays into a common array space. This mapping is done in such a way that the elements (from different arrays) that are accessed one after another in the original execution are stored in consecutive locations in the new array space. This helps to reduce inter-array conflict misses dramatically. We call the data transformations that work on multiple arrays simultaneously the *generalized data transformations*. This is in contrast to many previous compiler work on data transformations (e.g., [8, 6, 3]) that handle each array in isolation (that is, each array is transformed independently and the dimensionality of an array is not changed). Our approach is also different from the previously proposed array interleaving strategies (e.g., [4, 5]) in two aspects. First, our data transformations are more general than the classical transformations used for interleaving. Second, we present a strategy for selecting a dimensionality for the target common array space.

We present the theory behind the generalized data transformations in Section 3. Section 4 presents our overall approach. Our experimental results (Section 5) demonstrate that the generalized data transformations are very effective in improving data cache behavior of embedded applications. Finally, in Section 6, we summarize our contributions.

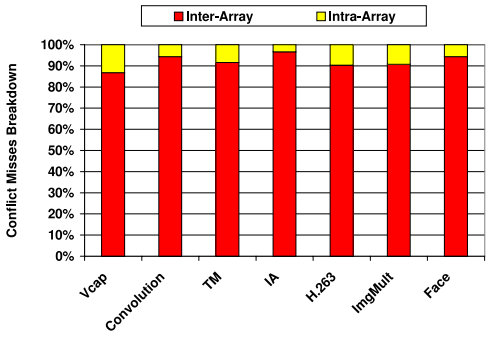

Figure 3. Contribution of inter-array and intra-array conflict misses.

2. Assumptions and Background

We consider the case of references (accesses) to arrays with affine subscript functions in nested loops, which are very common in array-intensive embedded image/video applications [2]. Let us consider such an access to an m-dimensional array in an n-deep loop nest. We use $\bar{I}$ to denote the iteration vector (consisting of loop indices starting from the outermost loop). Each array reference can be represented as $X\bar{I}+\bar{x}$, where the $m \times n$ matrix X is called the reference matrix [12] and the m-element vector $\bar{x}$ is called the offset vector. In order to illustrate the concept, let us consider an array reference $A(i+3, i+j-4)$ in a nest with two loops: i (the outer loop) and j (the inner loop). For this reference, we have

$$\bar{I} = \begin{pmatrix} i \\ j \end{pmatrix},\ X = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix},\ \text{and}\ \bar{x} = \begin{pmatrix} 3 \\ -4 \end{pmatrix}.$$

In execution of a given loop nest, an element is reused if it is accessed (read or written) more than once in a loop. There are two types of reuses: temporal and spatial. Temporal reuse occurs when two references (not necessarily distinct) access the same memory location; spatial reuse arises between two references that access nearby memory locations [13]. The notion of nearby locations is a function of the memory layout of elements and the cache topology. It is important to note that the most important reuses (whether temporal or spatial) are the ones exhibited by the innermost loop. If the innermost loop exhibits temporal reuse for a reference, then the element accessed by that reference can be kept in a register throughout the execution of the innermost loop (assuming that there is no aliasing). Similarly, spatial reuse is most beneficial when it occurs in the innermost loop because, in that case, it may enable unit-stride accesses to the consecutive locations in memory. It is also important to stress that reuse is an intrinsic property of a program. Whenever a data item is present in the cache at the time of reuse, we say that the reference to that item exhibits locality. The key optimization problem is then to convert patterns of reuse into locality, which depends on a number of factors such as cache size, associativity, and block replacement policy. Consider the reference $A(i+3, i+j-4)$ mentioned above in a nest with the outer loop i and the inner loop j. In this case, each iteration accesses a distinct element (from array A); thus, there is no temporal reuse. However, assuming a row-major memory layout, successive iterations

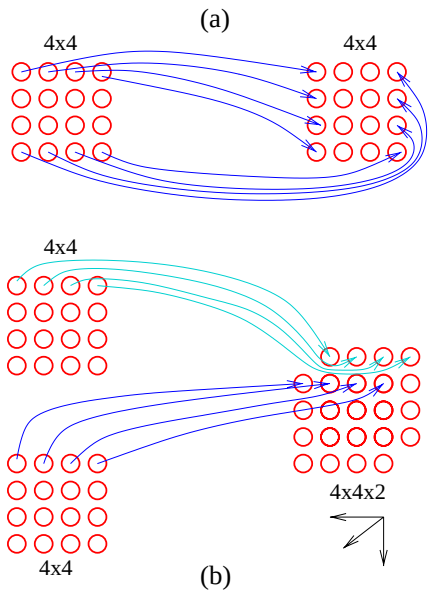

Figure 4. (a) 2D to 2D data mapping. (b) 2D to 3D generalized data mapping.

of the j loop (for a fixed value of i) access the neighboring elements on the same row. We express this by saying that the reference exhibits spatial reuse in the j loop. Since this loop (j loop) is the innermost loop, we can expect that this reuse will be converted into locality during execution. However, this would not be the case if i was the inner loop and j was the outer loop.

Consider the application of a non-singular linear loop transformation to an n-deep loop nest that accesses an array with the subscript function represented as $X\bar{I} + \bar{x}$. This transformation can be represented by an $n \times n$ non-singular transformation matrix T, and maps the iteration $\bar{I}$ of the original loop nest to the iteration $\bar{I'} = T\bar{I}$ of the transformed loop nest [12, 9, 13]. On applying such a transformation, the new subscript function is obtained as $XT^{-1}\bar{I'} + \bar{x}$; that is, the new (transformed) reference matrix is XT^{-1}. The loop transformation also affects the loop bounds of the iteration space that can be computed using techniques such as Fourier-Motzkin elimination [13]. In this paper, for convenience, we denote T^{-1} by Q. For example, a loop transformation such as

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

transforms an original loop iteration $(i, j)^T$ to $(j, i)^T$.

3. Generalized Data Transformations

Given an array reference $A(X\bar{I} + \bar{x})$ to an m-dimensional array A in an n-level nested loop, a data transformation transforms this reference to $A'(X'\bar{I} + \bar{x'})$. Here, A' corresponds to the transformed array, and X' and $\bar{x'}$ denote the transformed reference matrix and constant vector, respectively. In general, A' can have m' dimensions, where $m' \neq m$.

Formally, we can define a data transformation for an array A using a pair $(M_A, \bar{m_A})$. This pair (referred to as *data transformation* henceforth) performs the following two mappings:

$$\begin{aligned} X &\longrightarrow M_A X \\ \bar{x} &\longrightarrow M_A \bar{x} + \bar{m_A} \end{aligned}$$

In other words, the original reference matrix X is transformed to $M_A X$ and the original constant vector $\bar{x}$ is transformed to $M_A \bar{x} + \bar{m_A}$. We refer to M_A as the *data transformation matrix* and $\bar{m_A}$ as the *displacement vector*. If $A(X\bar{I} + \bar{x})$ is the original reference and $A'(X'\bar{I} + \bar{x'})$ is the transformed reference, we have $X' = M_A X$ and $\bar{x'} = M_A \bar{x} + \bar{m_A}$. Note that M_A is m'-by-m and $\bar{m_A}$ has m' entries. An example data transformation is depicted in Figure 4(a). In this data transformation, a 4×4 array (for illustrative purposes) is transposed; that is, the array element $(i, j)^T$ in the original array is mapped to array element $(j, i)^T$ in the transformed space. Note that in this case we have:

$$M_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{and} \quad \bar{m_A} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

3.1. Motivation

In many previous compiler-based studies that employ data transformations for improving cache behavior (e.g., [8, 10, 6]), only the cases where $m' = m$ are considered. This is because many of these studies target at reducing capacity misses, rather than conflict misses, and, focusing on each array individually is sufficient for reducing capacity misses in most cases. However, such an approach may not be very effective with conflict misses. Consider, for example, the following loop nest:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    k+= A(i,j) + B(i,j)
```

In this nest, assuming row-major memory layouts (as in C), both the references exhibit perfect spatial locality when considered in isolation. However, if the base addresses of arrays A and B happen to map on the same cache line, a miss rate of 100% is possible due to the data cache conflicts between these two array references (i.e., inter-array conflicts). A generalized data transformation, on the other hand, can prevent this by mapping them to the same array index space (data space). Consider, for example, the following data transformations (depicted in Figure 4(b) for m=2 and m'=3):

$$\begin{aligned} A(i,j) &\longrightarrow A'(i,j,0) \\ B(i,j) &\longrightarrow A'(i,j,1) \end{aligned}$$

After these transformations are applied, a given loop iteration $(i, j)^T$ accesses two consecutive elements from A' (the transformed array). Consequently, the chances for conflict misses are reduced significantly. Note that, in this specific example, we have

$$M_A = M_B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}; \bar{m_A} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \text{ and } \bar{m_B} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

In the next subsection, we formulate the problem of determining the data transformation matrices and displacement vectors for each array in a given application.

3.2. Problem Formulation and Solution

In this section, we formulate the problem at several levels of complexity, starting from the most simple case and ending with the most general case. Let us assume first that we have a single nest that accesses two different arrays using a single reference per array. Without loss of generality, we use $A(X_1\bar{I} + \bar{x_1})$ and $B(X_2\bar{I} + \bar{x_2})$ to refer to these references. Our task is to determine two data transformations $(M_A, \bar{m_A})$ (for array A) and $(M_B, \bar{m_B})$ (for array B) such that a given iteration accesses consecutive elements in the loop body.

In mathematical terms, since the transformed references are $M_A X_1 \bar{I} + M_A \bar{x_1} + \bar{m_A}$ and $M_B X_2 \bar{I} + M_B \bar{x_2} + \bar{m_B}$, for the best data locality, our data transformations should satisfy the following two constraints:

$$M_A X_1 = M_B X_2$$

$$(M_A \bar{x_1} + \bar{m_A}) - (M_B \bar{x_2} + \bar{m_B}) = (0, 0, ..., 0, 0, 1)^T.$$

The first constraint demands that the transformed reference matrices should be the same so that the difference between the memory locations accessed by these two references (in a given iteration) does not depend on a specific loop iteration value. This is important as if the difference between the accessed locations depends on loop iterator value, the locality becomes very difficult to exploit. The second constraint, on the other hand, indicates that the difference between the said memory locations should be 1. If we are able to satisfy these two constraints, this means that a given loop iteration accesses successive elements in memory; therefore, the possibility of cache conflict within a loop iteration is eliminated. One way of determining the data transformations $(M_A, \bar{m_A})$ and $(M_B, \bar{m_B})$ from these constraints is as follows. First, from the first constraint above, we can determine M_A and M_B. Then, substituting (the values of the elements of) these matrices in the second constraint, we can solve that constraint for $\bar{m_A}$ and $\bar{m_B}$. For example, in the loop nest above (in Section 3.1), this strategy determines the data transformations given earlier.

It should be stressed that this approach can improve capacity misses as well (in addition to conflict misses). This is because in determining the data transformation matrices (from the first constraint) we can improve self-spatial reuse too. In the following, we generalize our method to handle more realistic cases. We also present a general solution method that can be employed within an optimizing compiler that targets embedded environments where data space optimizations (minimizations) are critical.

Let us now focus on a single nest with multiple arrays. Assume further that each array is accessed through a single reference only. Without loss of generality, let $A_i(X_i\bar{I}+\bar{x_i})$ be the reference to array A_i, where $1 \leq i \leq s$. We also assume that the these references are touched (during program execution) in increasing values of i (i.e., from 1 to s). We want to determine data transformations $(M_{A_i}, \bar{m_{A_i}})$. We have two different sets of constraints (one for the data transformation matrices and the other for the displacement vectors):

$$M_{A_i} X_i = M_{A_j} X_j$$

$$(M_{A_{k+1}} \bar{x_{k+1}} + \bar{m_{A_{k+1}}}) - (M_{A_k} \bar{x_k} + \bar{m_{A_k}}) = (0, 0, ..., 0, 0, 1)^T,$$

where $1 \leq i, j \leq s$ and $1 \leq k \leq (s-1)$. Satisfying these constraints for all possible i, j, and k values guarantees good spatial locality during nest execution.

The discussion above assumes that each array has a single reference in the loop body. If we have multiple references to the same array, then the corresponding constraints should also be added to the constraints given above. Let us assume that there are t_i references to array A_i and that these references are accessed (during nest execution) in the order of 1,2,..,t_i. So, for a given array A_i, we have the following constraints:

$$M_{A_i} X_{i,j} = M_{A_i} X_{i,j'}$$

$$(M_{A_i} \bar{x_{i,(k+1)}} + \bar{m_{A_i}}) - (M_{A_i} \bar{x_{i,k}} + \bar{m_{A_i}}) = (0, 0, ..., 0, 0, 1)^T,$$

where $1 \leq j, j' \leq t_i$, $1 \leq k \leq (t_i - 1)$, and $X_{i,j}$ and $\bar{x_{i,j}}$ are the jth reference matrix and constant vector for array A_i, respectively.

It should be observed that the system of constraints given above may not always have a valid solution. This is because the second constraint above simplifies to $M_{A_i}(\bar{x_{i,(k+1)}} - \bar{x_{i,k}}) = (0, 0, ..., 0, 0, 1)^T$, and it may not be possible to find an M_{A_i} to satisfy both this and $M_{A_i} X_{i,j} = M_{A_i} X_{i,j'}$. We note, however, that if $X_{i,j} = X_{i,j'}$, we can drop $M_{A_i} X_{i,j} = M_{A_i} X_{i,j'}$ from consideration (as it is always satisfied). This case occurs very frequently in many embedded image processing applications (e.g., in many stencil-type computations). As an example, consider the references $A(i, j)$ and $A(i-1, j+1)$ in a nest with two loops, i and j. Since $\bar{x_{i,j}} - \bar{x_{i,j'}} = (1, -1)^T$, we need to select a M_{A_i} which satisfies the following constraint:

$$M_{A_i} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

It is easy to see that a possible solution is:

$$M_{A_i} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

The transformed references are then $A'(i+j, 0, i)$ and $A'(i+j, 0, i+1)$. We observe that for a given loop iteration $(i, j)^T$, these two references access consecutive memory locations.

So far, we have focussed only on a single nest. Many large array-intensive embedded applications consist of multiple nests. Our formulation above can easily extend to the cases where we have multiple nests in the application. First, if two nests of an application do not share any array between them, then there is no point in trying to handle these two nests together. In other words, in this case, each nest can be handled independently. On the other hand, if the nests share arrays, then the compiler needs to consider them together. In mathematical terms, the constraints (equations) given above should be written for each nest separately. Note, however, that the equations for nests that access the same array A should use the same data transformation matrix and

same displacement vector (in each nest). This is because in this paper we assign a single layout (i.e., a single data transformation) to each array; that is, we do not consider dynamic data transformations during the course of execution.

Given a program with multiple arrays referenced by multiple nest, the system of equations built (in terms of data transformation matrices and displacement vectors) may not have a solution. In this case, to find a solution, we need to drop some equations (constraints) from consideration. To select the equations to drop, we can rank the array references with respect to each other. More specifically, if an array reference is (expected to be) touched (during execution) more frequently than another reference, we can drop the latter from consideration and try to solve the resulting system again. We repeat this process until the resulting system has a solution. To determine the relative ranking of references, our current approach exploits profile information. More specifically, it runs the original program several times with typical input sets and records the number of times each array reference is used (touched). Then, based on these numbers, the references are ordered.

3.3. Dimensionality Selection

In our discussion so far, we have not explained how we determine the dimensionality of the target (common) array (data space). Consider the following nest which accesses four two-dimensional (2D) arrays.

$$\begin{array}{l} \text{for } (i = 0; i < n; i++) \\ \quad \text{for } (j = 0; j < n; j++) \\ \quad\quad k += A(i,j) + B(i,j) + C(i,j) + D(i,j) \end{array}$$

These arrays can be mapped to a common array space in multiple ways. Three such ways are given in Table 1. The second column shows the case where the target space (i.e., the common array space) is two-dimensional. Note that, in this case, the coefficient 4 (in front of loop index j) is necessary so that the elements from the different arrays can be interleaved. The next two columns illustrate the cases where the target space is three-dimensional and four-dimensional, respectively. While in these cases we do not have explicit coefficients (as in the 2D case), the address computation (that will be performed by the back-end compiler) is more involved as the number of dimensions is larger. It should be emphasized that all these three mapping strategies have one common characteristic: for a given loop iteration, when we move from one iteration to another, the array elements accessed are consecutively stored in memory (under a row-major memory layout). We also note that, for this example, going beyond four-dimensional case does not make much sense as some array elements would be unused. This is because we have only four references in the loop body.

In selecting the dimensionality of the common address space, our current implementation uses the following strategy. Let q be the number of references (to distinct arrays) in the loop body and m the dimensionality of the original arrays. We set the dimensionality of the common (array) space to m'=max$\{q, m\}$. This is reasonable, because as we mentioned above, there is no point in selecting a dimensionality which is larger than the number of references in the loop body. Also, we want the dimensionality of the common to be at least as large as the dimensionality of the original arrays.

Original	2D	3D	4D
$A(i,j)$	$A'(i,4j-3)$	$A'(i,j,0)$	$A'(i,j,0,0)$
$B(i,j)$	$A'(i,4j-2)$	$A'(i,j,1)$	$A'(i,j,0,1)$
$C(i,j)$	$A'(i,4j-1)$	$A'(i,j,2)$	$A'(i,j,1,0)$
$D(i,j)$	$A'(i,4j)$	$A'(i,j,3)$	$A'(i,j,1,1)$

Table 1. Three different mapping strategies.

4. Discussion and Overall Approach

So far, we have discussed the mechanics of the generalized data transformations and tried to make a case for them. As will be shown in the next section, the generalized data transformations can bring about significant performance benefits for array-intensive embedded applications. In this section, we discuss some of the critical implementation choices that we made and summarize our overall strategy for optimizing data locality.

An important decision that we need to make is to select the arrays that will be mapped onto a common address space. More formally, if we have s arrays in the application, we need to divide them into *groups*. These groups do not have to be disjoint (that is, they can share arrays); however, this can make the optimization process much harder. So, our curent implementation uses only disjoint groups. The question is to decide which arrays need to be placed into the same group. A closer look at this problem reveals that there are several factors that need to be considered. For example, the dimensionality of the arrays might be important. In general, it is very difficult (if not impossible) to place arrays with different dimensionalities into the common space. So, our current implementation requires that the arrays to be mapped together should have the same number of dimensions. However, not all the arrays with the same number of dimensions should be mapped together. In particular, if two arrays are not used together (e.g., in the same loop nest), there is no point in trying to map them to a common array space to improve locality. Therefore, our second criterion to select the arrays to be mapped together is that they should be used together in a given nest. To have a more precise measure, our current approach uses the following strategy. We first create an affinity graph such that the nodes represent arrays and the edges represent the fact that the corresponding arrays are used together. The edge weights give the number of loop iterations (considering all the loop nests in the code) that the corresponding arrays are used together. Then, we cluster the nodes in such a way that the nodes in a given cluster exhibit high affinity (that is, they are generally used together in nests). Each cluster then corresponds to a group provided that the nodes in it also satisfy the other criteria. The last criterion we consider is that the arrays in the same group should be accessed with the same frequency. For example, if one array is accessed by the innermost loop whereas the other array is not, there is not much point in trying to place these arrays into a common address space.

Another important decision is to determine the mapping of the arrays in a given group to the corresponding common address space. To do this, we use the affinity graph discussed in the previous paragraph. If the weight of an edge in this graph is very high, that means it is beneficial to map the corresponding arrays in such a way that the accesses to them

occur one after another. As an example, suppose that we have three two-dimensional arrays are to be mapped onto the same address space. Suppose also that the common address space is three-dimensional. If two of these arrays are used together more frequently, then, if possible, their references should be mapped to $A'(i,j,0)$ and $A'(i,j,1)$, repsectively (or to $A'(i,j,1)$ and $A'(i,j,2)$); that is, they should be consecutive in memory.

Based on this discussion, our overall optimization strategy works as follows. First, we build the affinity graph and determine the groups taking into account the dimensionality of the arrays and access frequency. Then, we use the strategy discussed in the previous two sections in detail to build a system of equations. After that, we try to solve this system. If the system has no solution, we drop some equations from consideration (as explained earlier), and try to solve the resulting system again. This iterative process continues until we find a solution. Then, the output code is generated. The code generation techniques for data transformations are quite well-known and can be found elsewhere [8, 3].

5. Experimental Setup and Results

All results presented in this section are obtained using an in-house execution-driven simulator that simulates an embedded MIPS processor core (5Kf). 5Kf is a synthesizable 64-bit processor core with an integrated FPU. It has a six-stage, high-performance integer pipeline optimized for SoC design that allows most instructions to execute in 1 cycle and individually configurable instruction and data caches. The default configuration contains separate 8KB, direct-mapped instruction and data caches. Both the caches have a (default) line (block) size of 32 bytes. In all experiments, a miss penalty of 70 cycles is assumed. Our simulator takes a C code as input, simulates its execution, and produces statistics including cache hit/miss behavior and execution time.

To measure the effectiveness of our approach, we used seven array-intensive embedded applications from image and video processing domain. `Vcap` is a video capture and processing application. It generates video streams of different picture sizes, color spaces, and frame rates. `Convolution` is an efficient implementation of a convolution filter. One of the abilities of this benchmark is that it can process data elements of variable lengths. `TM` is an image conversion program that converts images from TIFF to MODCA or vice versa. `IA` is an image understanding code that performs target detection and classification. `H.263` is a key routine from a simple H.263 decoder implementation. `ImgMult` is a subprogram that multiplies three images with each other, and adds the resulting images and one of the input images. `Face` is a face recognition algorithm. More details about these applications are beyond the scope of this paper.

Figure 5 gives the percentage reduction in cache misses (as compared to the original codes) when the global data transformations are used (without the support of loop transformations). Each bar in this graph is broken into two portions to show the reductions in conflict misses and other misses separately. We see from these results that overall (when considered all types of cache misses) we have nearly a 53% reduction in misses. We also observe that more than half of this reduction occur in conflict misses, clearly demonstrating the effectiveness of our approach in reducing conflict misses.

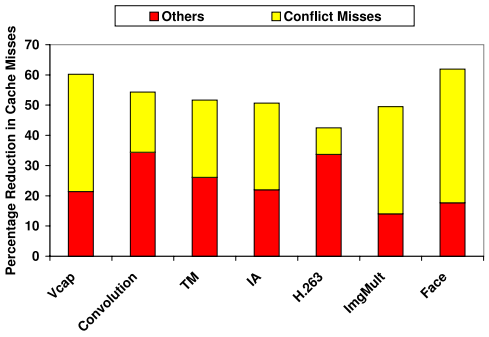

Figure 5. Reduction in cache misses.

It is also important to compare these results with those of array-padding presented in Figure 2. Our generalized data transformations bring much more significant reductions in cache misses as compared to array padding. More importantly, these benefits are obtained without increasing the data space requirements of applications.

6. Concluding Remarks

Most existing work based on data transformations for improving data cache behavior of array dominated codes target a single array at a time. In this work, we develop a new, generalized data space transformation theory and present experimental data to demonstrate its effectiveness. Our results clearly indicate that the proposed approach is very successful in practice.

References

[1] S. P. Amarasinghe et al. The SUIF compiler for scalable parallel machines. In *Proc. the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[2] F. Catthoor et al. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[3] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. Programming Language Design and Implementation*, pp. 205–217, 1995.

[4] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*, Ph.D Thesis, Rice University, Houston, Texas, January 2000.

[5] M. Kandemir. Array unification: a locality optimization technique. In *Proc. International Conference on Compiler Construction*, ETAPS'2001, April, 2001.

[6] M. Kandemir et al. Improving locality using loop and data transformations in an integrated framework. In *Proc. International Symposium on Microarchitecture*, Dallas, TX, December, 1998.

[7] I. Kodukula et al. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[8] 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.

[9] W. Li. *Compiling for NUMA parallel machines*. Ph.D. Thesis, Cornell University, Ithaca, New York, 1993.

[10] 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.

[11] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proc. the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[12] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30–44, June 1991.

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