A Compiler-Based Approach for Improving Intra-Iteration Data Reuse*

Mahmut Kandemir
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, 16802, USA

Abstract

Intra-iteration data reuse occurs when multiple array references exhibit data reuse in a single loop iteration. An optimizing compiler can exploit this reuse by clustering (in the loop body) array references with data reuse as much as possible. This reduces the number of intervening references between references to the same array and improves overall execution time and energy consumption. In this paper, we present a strategy where inter-statement and intra-statement optimizations are used in concert for optimizing intra-iteration data reuse. The objective is to cluster (within the loop body) the array references with spatial or temporal reuse. Using four array-intensive applications from image processing domain, we show that our approach improves the cache behavior of programs by 13.8% on the average.

1 Introduction and Motivation

The performance of data-intensive applications from embedded image and video processing domains is greatly affected by data locality behavior [1]. Most of these applications exhibit regular data access patterns which can be improved using compiler-directed loop (iteration space) and array layout (data space) optimizations. The objective of these optimizations is to maximize spatial and temporal data reuse in the inner loop positions. Loop transformations achieve this by modifying the execution order of loop iterations. Among commonly-used loop optimizations are unimodular transformations (loop permutation, loop skewing, and loop reversal), iteration space tiling, and loop fusion [9]. Data transformations, on the other hand, do not modify loop execution order; instead, they transform memory layout of a given multi-dimensional array from one form to another [5, 2]. An example would be transforming memory layout of a two-dimensional array from row-major to column-major if doing so improves spatial locality. Loop and data transformation have advantages and disadvantages with respect to each other; [6] shows that there are programs that can be optimized best using a mix of loop and data transformations.

An important characteristic of most of the previous loop-based techniques is that when they evaluate data locality, they focus on successive loop iterations and observe how a given array is accessed by these iterations. For example, assume a code fragment which consists of a two-level nested loop (i being the outer loop and j being the inner loop) and contains a reference such as $U[j][i]$ to a two-dimensional row-major array. In evaluating locality of this reference, previous loop-based techniques consider two successive iterations of the innermost loop. Since for a fixed value of the i loop, two successive iterations of the j loop access two elements from different rows, many loop-based optimization frameworks decide that this array reference is not good from the data locality viewpoint. One of the most commonly used techniques for improving locality behavior of a reference like this is to interchange the order of loops; that is, making the i loop the inner loop. After this transformation, for a fixed value of the j loop (now in the outer position), successive iterations of the i loop (now in the inner position) access consecutively-stored elements in memory. In most cases, such a transformation improves cache locality dramatically. Since such an approach bases its transformation decision on inter-iteration access pattern (i.e., the access pattern exhibited by two successive inner loop iterations), we refer to it as *inter-iteration data reuse optimization.* Many locality-oriented loop-level optimizations proposed in the literature fall into this category.

In many array-intensive codes, however, there are also numerous opportunities to take advantage of *intra-iteration data reuse.* An intra-iteration data reuse is said to occur when there are multiple references to the same array within the loop and the execution of a *single loop iteration* leads to a data reuse among these references. For instance, assuming a loop with an index i, two references $V[i]$ and $V[i+2]$ exhibit an intra-iteration data reuse. This is because for a given iteration i, the two elements accessed through these two references are only two locations apart in memory. Unless the cache line size is smaller than three array elements or these two elements happen to map onto a cache line boundary, these two accesses will map to the same cache line.

*This work was supported in part by NSF awards #0093082 and #0103583.

In this case, even if the first reference experiences a cache miss, the second reference will result in a hit. It should be noted, however, it may not always be easy to exploit intra-iteration reuse. Consider, for example, a statement such as $U[i] = W[i-1] + U[i+1] + V[i-2] + W[i+1]$ in a loop i. Here, we have two intra-iteration data reuse opportunities: one due to references $W[i-1]$ and $W[i+1]$ and the other due to $U[i+1]$ and $U[i]$. However, between accesses $W[i-1]$ and $W[i+1]$ there are two other array references. It is possible that one of these references can cause the cache line holding $W[i-1]$ and $W[i+1]$ to be displaced from the cache before the reuse occurs. If this is the case, then the second access to array W will bring the cache line in question from main memory to cache again. A similar scenario occurs with references $U[i+1]$ and $U[i]$ as well. This discussion assumes that the order of accesses for these references (for a given iteration i) is $W[i-1]$, $U[i+1]$, $V[i-2]$, $W[i+1]$, and $U[i]$; i.e., the usual expression evaluation order in structural languages such as C. On the other hand, if we re-write this statement as $U[i] = W[i-1] + W[i+1] + V[i-2] + U[i+1]$, we generate an access sequence of $W[i-1]$, $W[i+1]$, $V[i-2]$, $U[i+1]$, and $U[i]$. Note that in this new access sequence, the accesses to array W come one after another. A similar argument holds for references to array U. We conclude from this small example that *intra-statement transformations* can be useful in exploiting intra-iteration data reuse.

Let us now consider a loop nest that contains the following four statements in that order: $U[i] = U[i-1]+U[i+1]$; $V[i] = V[i-2]-1$; $U[i] = U[i] + U[i-2] + U[i+2]$; and $k = k + V[i] + V[i+1] + V[i+2]$. It should be noted that there is a significant amount of reuse between references to array U (and similarly between references to array V). However, it is not clear much of this data reuse will be exploited at runtime (that is, it would be converted to locality) as some references to the same array are a statement apart from each other. If the compiler re-orders these statements in the loop body as $U[i] = U[i-1] + U[i+1]$; $U[i] = U[i]+U[i-2]+U[i+2]$; $V[i] = V[i-2]-1$; and $k = k+V[i]+V[i+1]+V[i+2]$, the chances for exploiting reuse become much higher. This is because in this transformed version, the statements that access the same array are clustered together. This example illustrates that *inter-statement transformations* can also be useful in exploiting intra-iteration data reuse.

It should be mentioned that some of the problems regarding not being able to exploit intra-iteration reuse can be alleviated using array padding [7], a compiler optimization that reduces the number of conflict misses by padding array dimensions (called intra-array padding) and/or augmenting the code by adding dummy arrays (called inter-array padding). However, it may not be possible to use array padding to solve this problem entirely. First, there are some codes which cannot be optimized using array padding. Second, array padding in general increases the data space requirements of the code. This may not be acceptable in an embedded processing environment where the size of data memory is customized according to the size of data space requirements of the application. That is, since the size of data space directly determines the size of the physical memory, it is very important to minimize the memory requirements of the application as much as possible. As opposed to array padding, our approach does not increase data space requirements; thus, it is more suitable for embedded environments that process data-intensive applications.

In this paper, we present a strategy where inter-statement and intra-statement optimizations are used in concert for optimizing intra-iteration data reuse. The objective is to cluster (within the loop body) the array references with spatial or temporal reuse. Using four array-intensive applications, we show that this approach improves the cache behavior of our applications by 18.3% on an average.

The rest of this paper is organized as follows. Section 2 presents our representation of a nested loop that references a number of arrays. This is the main representation that our optimization strategy operates on. Section 3 gives the details of our optimization algorithm. Section 4 introduces our experimental setup and presents preliminary results. Section 5 gives a summary of major contributions of this work.

2 Code Representation

We represent a given nested loop using a *Reuse Graph* (RG). A reuse graph is an undirected graph $RG(\mathcal{V}, \mathcal{E})$ where each $v_i \in \mathcal{V}$ represents an array reference in the code and each $e = (v_i, v_j) \in \mathcal{E}$ indicates that two references represented by v_i and v_j exhibit intra-iteration data reuse. In order for two references represented by v_i and v_j to have intra-iteration data reuse, the following three conditions should be met. First, the two references should belong to the same array. Second, these references should belong to the same uniformly generated reference (UGR) set.[1] Third, the memory locations accessed by these references should be within the size of the cache line. That is, if these references access memory locations m_i and m_j, then the condition $|m_i - m_j| < l$ should be satisfied, where l is the cache line (block) size. As an example, assuming a cache line size of 4 elements, row-major memory layouts, and a nest with two loops (i and j), the references $U[i][j]$ and $U[i][j+1]$ exhibit intra-iteration data reuse. On the other hand, under the same assumptions, $V[i]$ and $V[i+5]$, $W[i]$ and $W[2i]$, and $X[i-1][j]$ and $X[j][i-1]$ do not exhibit intra-iteration data reuse.

A reuse graph represents all intra-iteration reuse available in the nest. Whether this reuse can be converted into locality or not depends largely on the flexibility we have

[1] Two array references are said to be uniformly generated (or to belong to the same uniformly-generated reference set) if they have the same subscript expressions in each array dimension (array index position) up to constant term(s) [3]. For example, $U[i-1][j+2]$ and $U[i+3][j]$ are uniformly-generated references, whereas $W[i][j]$ and $W[j][i]$ are not.

in ordering the accesses to the references in the code. The purpose of our optimization strategy is to convert as many data reuses in the nest as possible to locality. We say that a data reuse is converted to locality (or it is exploited) if the reused item is found in the cache. We attempt to achieve this by clustering (bringing together) the references to the same array as much as possible. Note that this involves modifying the access order of array references in the nest body. Obviously, any such modification is restricted by intrinsic data dependences in the code and by the expression evaluation rules of the language used. In order to keep the problem manageable, we make the following assumptions in this work: (i) we assume that no source-level statement is broken into multiple sub-statements; and (ii) we assume that back-end optimizations or the underlying hardware do not modify the execution order of load/store operations dictated by the expression evaluation rules of the high-level language used. The reason for the first restriction is that dividing a given statement into multiple sub-statements in general entails the use of a temporary array. This, in turn, increases the data space requirements of the application, which is not a desirable approach for many embedded systems. Similarly, we do not perform any statement-level transformation that increases the number of array references in the nest. The second restriction is also reasonable for embedded systems. In general, the microprocessors used in embedded systems are not as powerful as their general-purpose counterparts. Consequently, many of them do not have hardware that speculatively re-orders loads and stores. Also, it is possible to modify the back-end instruction scheduler such that instruction movements are allowed only for non-memory operations. Based on this discussion, in the rest of this paper, we assume that in a given expression that involves array references, the references are touched from left to right, the * and / operations having higher precedence than + and - operations as usual. Obviously, these precedence relations can be overwritten using parentheses.

Figure 1(a) gives the RG for the following code fragment:

$$
\begin{array}{l}
for(i = 1; i \leq N-1; i++) \\
\{ \\
\quad U[i] = V[i] - 1; \\
\quad W[i] = W[i-1] + X[i] + W[i] + X[i+1]; \\
\quad V[i] = V[i-1] + U[i+1]; \\
\quad X[i] = (X[i+1] - W[i-1])/2; \\
\}
\end{array}
$$

Each edge in this graph (called *reuse edge*) indicates that the connected references should be accessed successively after the transformation. If, after the transformation, the two references connected by an edge in the reuse graph are accessed successively, we say that the corresponding reuse is satisfied (i.e., the reuse between the two references in question is exploited). Obviously, in many cases, it may not be possible to transform the code such that all reuses in the code are satisfied. Instead, the objective of our optimization strategy is to maximize the number of satisfied reuses in a given loop body. We try to achieve this by using a mix of intra-statement and inter-statement transformations. It should be emphasized that even if two references to the same array are exactly the same (i.e., they have the same subscript expressions in each array dimension), we still need to try to bring them together as they might be far apart in the loop body. Note that the references in this example are touched (in execution) in the following order:

$$V[i], U[i], W[i-1], X[i], W[i], X[i+1], W[i], V[i-1], U[i+1],$$
$$V[i], X[i+1], W[i-1], X[i].$$

In the remainder of this paper, we refer to such a reference sequence as *access pattern* or *access order.* In this access

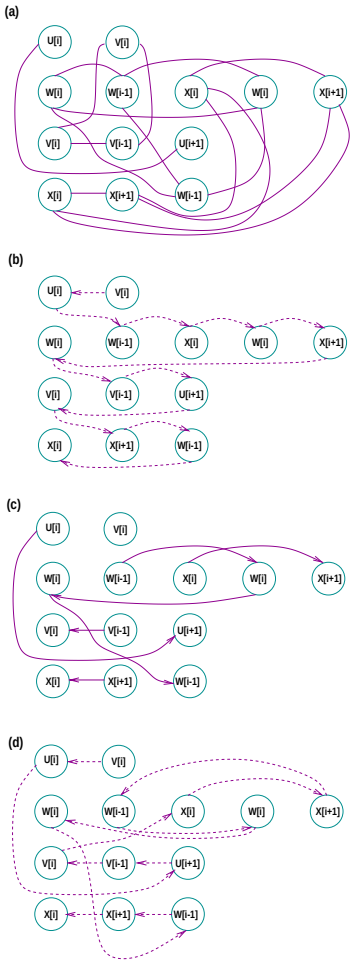

Figure 1. (a) An example reuse graph (RG). (b) Default execution graph. (c) Different paths in the reduced reuse graph (RRG). (d) Optimized execution graph.

pattern, there are total 12 transitions between references. It should be noted, however, that no transition in this pattern exhibits data locality; that is, no two array references that exhibit data reuse are accessed in sequence. In the rest of the paper, a transition, say, from a reference $W[i-1]$ to a reference $X[i]$ is denoted using $W[i-1] \rightarrow X[i]$. It should also be noted that the loop body itself contains many data reuses. Our optimization strategy detailed in the next section transforms this code to the following code:

$$
\begin{array}{l}
for(i=1; i \leq N-1; i++) \\
\{ \\
\quad U[i] = V[i] - 1; \\
\quad V[i] = U[i+1] + V[i-1]; \\
\quad W[i] = X[i] + X[i+1] + W[i-1] + W[i]; \\
\quad X[i] = (W[i-1] - X[i+1])/2; \\
\}
\end{array}
$$

Now, the new access pattern is:

$$V[i], U[i], U[i+1], V[i-1], V[i], X[i], X[i+1], W[i-1], W[i],$$
$$W[i], W[i-1], X[i+1], X[i].$$

In this transformed access pattern, 7 out of 12 transitions exhibit locality. These transitions are $U[i] \rightarrow U[i+1]$; $V[i-1] \rightarrow V[i]$; $X[i] \rightarrow X[i+1]$; $W[i-1] \rightarrow W[i]$; $W[i] \rightarrow W[i]$; $W[i] \rightarrow W[i-1]$; and $X[i+1] \rightarrow X[i]$. The next section explains how we obtain this optimized code from the original one.

It should be emphasized that intra-iteration data reuse might come in forms that are quite different from traditional reuse. For example, assume a two-level nested loop, where i is the outer loop and j is the inner loop, that contains references $U[j][i]$, $U[j][i-1]$, $U[j][i-2]$, and $U[j][i-3]$ to a two-dimensional row-major array U. Considering each reference in isolation, a classical optimization algorithm can decide that no references in this code exhibit data reuse. This is because for a fixed value of i, the successive iterations of loop j access different rows of the array. However, if we focus on a single iteration only, these references exhibit intra-iteration data reuse assuming a cache line size of four array elements. In other words, even if the successive iterations of inner loop does not take advantage of a cache line, execution of a given loop iteration does. Now, supposing that we are not able to apply loop interchange to this code for improving its cache behavior, we can still take advantage of intra-iteration reuse.

3 Optimization Algorithm

Our starting point is a reuse graph such as the one in Figure 1(a). Using the nodes in the reuse graph we also define an execution graph $EG(\mathcal{V}', \mathcal{E}')$ which shows the (reference) access pattern in the loop body. For a given $RG(\mathcal{V}, \mathcal{E})$, the nodes in the corresponding $EG(\mathcal{V}', \mathcal{E}')$ are the same nodes in RG; that is, $\mathcal{V}' = \mathcal{V}$. An edge $e' = (v_i', v_j') \in \mathcal{E}'$ between two nodes $v_i', v_j' \in \mathcal{V}'$ indicates that the reference represented by v_i' is touched just before the reference represented by v_j' during execution. The graph in Figure 1(b) shows the original (unoptimized) execution graph for our example.

Let us now compare the graps in Figures 1(a) and (b). Recall that we indicated in the previous section that the original code fragment in our example does not exhibit any locality. A comparison of these two graphs gives us a clue as to why this is so. Each edge in Figure 1(a) indicates that (for exploiting data reuse) the associated references (nodes) should be brought together (i.e., in execution, they should be touched one after another). Ideally, we should have the same edge in the execution graph. Put it another way, if we have an edge between two nodes in the reuse graph and do not have the corresponding edge in the execution graph, this means that we are not exploiting the corresponding data reuse; that is, we are not converting the data reuse into locality. Then, we can re-express the problem of intra-iteration data reuse optimization as one of

> transforming loop body such that the number of edges in the reuse graph with corresponding edges in the execution graph will be maximized.

We say that an edge in the reuse graph is satisfied if there is a corresponding edge in the execution graph. There might be several reasons for an edge not being satisfied. For example, the expression evaluation rules of the language may prevent two variables (that are connected by a reuse edge) from being brought together. Data dependences in the code may also prevent two statements from being brought together. Our optimization strategy tries to derive an execution graph from a given reuse graph such that the number of satisfied edges is maximum and all data dependences and expression evaluation rules are preserved.

We first divide a given reuse graph into levels. Each level corresponds to a statement in the source code. As an example, the first level (level one) in Figure 1(a) contains the nodes $U[i]$ and $V[i]$, whereas the second level (level two) has the nodes $W[i]$, $W[i-1]$, $X[i]$, $W[i]$, and $X[i+1]$. For a given statement (level), we also mark each reference as either left-hand-side reference (or LHS node) or right-hand-side reference (or RHS node). In our example, in the statement in the third level $V[i]$ is a LHS reference whereas $V[i-1]$ and $U[i+1]$ are RHS references. Then, we eliminate some edges from the reuse graph if we detect that there is no way of satisfying them. For example, since we assume that no statement in the original code will be divided into multiple sub-statements, there is no way to satisfy a reuse edge from a RHS reference in level j to a RHS reference in level k, where $k \neq j$. Similarly, an edge from a LHS reference v_i to another LHS reference v_j cannot be satisfied unless there is no RHS reference in the level that v_j belongs to. In addition, an edge from a RHS reference to a LHS reference in another level should be dropped from consideration.[2] Finally, a reuse edge from v_i to v_j is eliminated

[2] However, this edge may still need to be maintained if it is possible to

if satisfying that edge would violate a data dependence or an expression evaluation rule. These four reasons, namely, (i) an edge from a RHS reference to another RHS reference in a different statement, (ii) an edge from a LHS reference to another LHS reference in a statement with a non-empty RHS, (iii) an edge from a RHS reference to a LHS reference in a different level, and (iv) an edge that violates a data dependence, may significantly reduce the number of edges to be considered in the rest of the optimization process. In our example reuse graph in Figure 1(a), the edge from $X[i]$ in level two to $X[i+1]$ in level four is eliminated due to reason (i). Similarly, the edge from $V[i]$ in level one to $V[i]$ in level three is eliminated due to reason (iii). The edge from $X[i]$ in level four to $X[i]$ in level two is eliminated as taking this edge would break a data dependence (reason (iv)).

After eliminating as many edges as possible from the reuse graph, the rest of our approach works on this reduced reuse graph (RRG). In a sense the edges in a given RRG represent *exploitable data reuses*, that is, the reuses that can actually be converted into locality. Once an RRG has been built, our approach proceeds as follows. It first determines all paths in the RRG. Note that a path in RRG is important as it represents a reuse chain; that is, a sequence of references with data reuse. Given a path, if it is possible to transform the code (i.e., to come up with an execution graph) such that all reuses in a given path are satisfied, then we can expect a good cache behavior at runtime. There are two potential problems here. First, it may not be possible to satisfy all reuses in a given path. Given a path, our approach tries to satisfy as many reuses as possible. Second, there might be multiple paths in the RRG. Therefore, we need to prioritize paths; in other words, we need to determine an order for processing paths. Our current implementation orders the paths according to their lengths; that is, the number of reuse edges they contain, and starts processing them with the longest path. The reason for this is that longer the path, more potential reuse it represents.

When analyzing paths, our approach also builds the optimized execution graph. More specifically, in processing a path, our approach performs all intra-statement and inter-statement transformations to maximize the number of satisfied reuse edges. To illustrate what kinds of transformations might be necessary under different circumstances, let us consider the RRG fragment depicted in Figure 2. This figure also shows two separate paths. The first path contains the edges (I), (II), (III), and (IV), whereas the second path contains the edge (V). Our approach first considers the first path as it is longer than the other. Let us first focus on the edge (I). Satisfying this edge means that the third statement should be the statement that immediately follows the first statement. Note that this implies an inter-statement transformation. Satisfying this edge also implies that the node $U8$ should be the first reference (that is, the leftmost one) accessed on the RHS of the statement. Note that this indicates

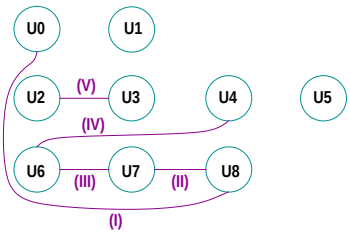

Figure 2. A reuse graph fragment and two paths.

an intra-statement transformation. Considering reuse edge (II), we can see that satisfying this edge requires that nodes $U7$ and $U8$ should be accessed one after another. Combining this requirement with that coming from edge (I), we see that variables $U8$ and $U7$ should be accessed in this order on the RHS of the original statement three. Consequently, the reuse edge (III) does not carry any additional information. This is because what this edge implies that $U7$ should be the last variable accessed on the RHS, but this fact has already been deduced from edges (I) and (II). Satisfying edge (IV) implies that $U4$ should be the first variable accessed on the RHS of the second statement and that this statement should immediately follow the original third statement. To summarize, by considering the first path in the figure, we have decided that the third statement should immeditely follow the first statement and the second statement should follow the third statement, determined the complete reference access pattern for the third statement, and concluded that $U4$ should be the first reference accessed in the second (original order) statement. Next, we move to analyzing the second path. This path contains only a single edge (edge (V)), and satisfying it means that $U3$ should be the last variable accessed on the RHS of this statement. Combining this information with the information coming from analyzing edge (IV), we can conclude that the variable access order on the RHS of the original second statement should be $U4$, $U5$, $U3$ (from left to right), as opposed to the original order $U3$, $U4$, $U5$. Note that this implies an intra-statement transformation; more specifically, a combination of commutativity and associativity transformations.

However, even considering only the exploitable reuses, we may not be able to satisfy all of them. This is because satisfying an exploitable data reuse may prevent another exploitable reuse from being satisfied. Our approach processes the edges in a given path one-by-one. When it faces a conflict, it just omits the edge that creates the conflict, and continues with the rest of the path. We have found that this approach is easy to implement and performs very well in practice. Returning to our example in Figure 1, Figure 1(c) shows the paths in the RRG. Figure 1(d), on the other hand, illustrates the optimized execution graph.

interpret it as an edge from the LHS reference to the RHS reference.

Benchmark Name	Total Reuses		Satisfied Reuses	
	Original	C-Opt	I-Opt	IC-Opt
wave	78	94	24 (31%)	31 (33%)
splat	123	161	71 (58%)	97 (60%)
3D	56	69	12 (21%)	15 (22%)
dfe	109	138	27 (24%)	43 (31%)

Figure 3. Total and satisfied intra-iteration data reuses in unoptimized and optimized codes.

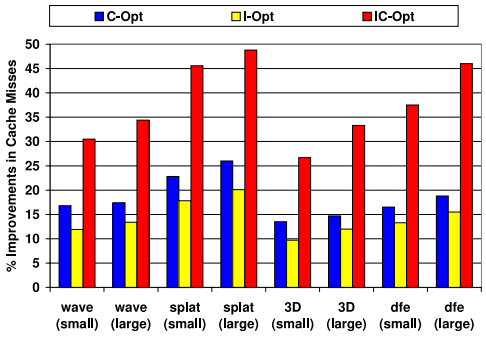

Figure 4. Percentage improvements in cache misses (with respect to Original).

4 Experimental Setup and Results

We tested the effectiveness of our algorithm using four array-intensive programs from the image processing domain: `wave`, `splat`, `3D`, and `dfe`. `wave` is a wavelet compression code that targets specifically medical applications. `splat` is a volume rendering application which is used in multi-resolution volume visualization through hierarchical wavelet splatting. `3D` is an image-based modeling application that simplifies the task of building 3D models and scenes. Finally, `dfe` is a digital image filtering and enhancement code. These C programs are written so that they can operate on images of different sizes. All results are obtained assuming an 8KB data cache with a hit latency of 2 cycles and a miss latency of 75 cycles.

We generated four different versions of each code in our experimental suite:

- `Original:` This is the original (unoptimized) code.
- `C-Opt:` This is a classical data locality optimization approach based on iteration and data space transformations. Specifically, for each nest, it first uses aggressive loop transformations to optimize temporal locality; and then, for array references without temporal locality, it uses data transformations (layout modifications) for optimizing their spatial locality. A detailed description of this approach is outside the scope of this paper and can be found elsewhere [4].
- `I-Opt:` This is the intra-iteration data reuse optimization approach presented in this paper. It does not apply any other loop or data transformation for enhancing cache behavior. Note that comparing the performance of this version with that of the Original version shows us how effective our approach is in optimizing intra-iteration reuse in unoptimized codes.
- `IC-Opt:` This is a strategy that combines the C-Opt and I-Opt. In our current implementation, it first applies C-Opt and then uses I-Opt to improve cache locality further. Note that comparing the performance of this version with that of C-Opt indicates how effective our approach is in optimizing intra-iteration reuse in optimized codes.

All results presented in the rest of this paper are *percentage improvements* over the Original version. For each code, we performed experiments with two different input sizes (total size of image data in the code): small and large. The small input sizes for `wave`, `splat`, `3D`, and `dfe` are 560 KB, 1.4MB, 2.2MB, and 882KB, respectively. The corresponding values for the large inputs are 1.4MB, 2.6MB, 3.0MB, and 1.2MB in that order.

Before presenting cache miss and execution time results, we give in Figure 3 the number of data reuses in each version of each code, assuming that a cache line can hold eight array elements. The values in columns two and three give the number of data reuses in the Original and C-Opt versions, respectively. The values in the fourth and fifth columns, on the other hand, show the number of reuses satisfied when I-Opt and IC-Opt versions are used. The values in parantheses give the same values as fractions of the corresponding values in the second and third columns. It can be observed from this table that, in both the cases, using our optimizarion strategy satisfies a large percentage of intra-iteration reuses. Although these results (static measures) are encouraging, it is also important to measure the impact of our approach on dynamic quantities such as cache misses and execution times.

Figure 4 gives improvements in data cache misses with respect to the Original version. We see that on the average C-Opt, I-Opt, and IC-Opt reduce cache misses by 18.3%, 13.8%, and 37.8%, respectively. From these results, we can make the following observations. First, intra-iteration reuse optimization strategy is very successful (even if it is not accompanied by any other locality optimization scheme) and generates reductions in the number of misses. Second, our approach is even more effective in optimizing locality of already optimized codes. This is because the optimized codes have more reuse edges and consequently present more opportunities for our intra-iteration optimization strategy. Third, we observe that the effectiveness of our strategy increases with the increased input size as larger image arrays place more pressure on data cache and create more optimization opportunities. Figure 5 gives the corre-

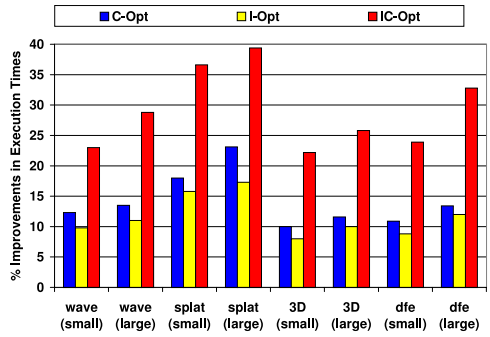

Figure 5. Percentage improvements in execution times (with respect to Original).

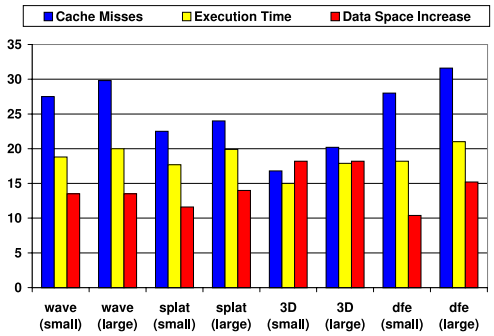

Figure 6. Array padding results.

sponding (percentage) improvements in execution times. It can be seen that the trends in this figure are very similar to those given for cache misses, indicating that cache locality is the primary factor determining the execution efficiency of these array-dominated codes. The average (across all codes) improvements due to C-Opt, I-Opt, and IC-Opt versions are 13.1%, 10.3%, and 28.1%, respectively. Among our codes, only `splat` is able to take advantage of loop fusion. The values reported above for this benchmark when the IC-Opt version is used has been obtained using a straightforward scheme where the compiler first uses loop fusion (along with other optimizations) and then utilizes our intra-iteration optimization algorithm.

We now compare our approach to array padding. The first two bars for each code in Figure 6 give the percentage improvements in both the cache misses and execution times (with respect to Original) when array padding is used in conjunction with the C-Opt version. We observe that the average improvements in cache misses and execution times are 25.0% and 17.5%, respectively. Note that these values are lower than the corresponding values for the IC-Opt version, showing that our strategy is more successful than array padding in eliminating conflict misses. In addition, array padding has an important drawback as far as embedded systems are concerned. It can significantly increase the overall data requirements of the code. The last bar (for each code) in Figure 6 shows this as percentage increase with respect to the IC-Opt version. We can see that the average increase in data space size is 14.3%. This is not a desirable result for embedded systems as application data requirements are typically the prime factor which determine the size of the physical memory. That is, an increase in data space size reflects on as an increase in area cost and dollar amount.

5 Conclusions

This paper presents a strategy for improving data reuse in executing a single iteration of a given loop nest. Our approach represents the potential intra-iteration data reuse in loop body as a reuse graph and determines the order in which the array references should be touched using a strategy based on path detection. Our preliminary results using a set of array-intensive image processing applications indicate that our approach is very successful in practice. In particular, when used in conjunction with classical locality-oriented compiler optimizations, it was able to improve overall execution time by as much as 39%.

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. *the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[3] 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.

[4] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In Proc. *International Symposium on Microarchitecture*, Dallas, TX, December, 1998.

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

[6] M. F. P. O'Boyle and P. M. W. Knijnenberg. Integrating loop and data transformations for global optimisation. In Proc. *Parallel Architectures and Compilation Techniques*. IEEE Press, October 1998.

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

[8] W. Tang, A. V. Veidenbaum, A. Nicolau, and R. Gupta. Conflict miss elimination by time-stride prefetch. *Technical Report ICS-TR-00-15*, UCI, March 200.

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