Compiler-Directed Selection of Dynamic Memory Layouts

Mahmut Kandemir and Ismail Kadayif
Micorsystems Design Laboratory
Pennsylvania State University
University Park, PA 16802, USA
{kandemir,kadayif}@cse.psu.edu

Abstract. Compiler technology is becoming a key component in the design of embedded systems, mostly due to increasing participation of software in the design process. Meeting system-level objectives usually requires flexible and retargetable compiler optimizations that can be ported across a wide variety of architectures. In particular, source-level compiler optimizations aiming at increasing locality of data accesses are expected to improve the quality of the generated code. Previous compiler-based approaches to improving locality have mainly focused on determining optimal memory layouts that remain in effect for the entire execution of an application. For large embedded codes, however, such static layouts may be insufficient to obtain acceptable performance. The selection of memory layouts that dynamically change over the course of a program’s execution adds another dimension to data locality optimization. This paper presents a technique that can be used to automatically determine which layouts are most beneficial over specific regions of a program while taking into account the added overhead of dynamic (runtime) layout changes. The results obtained using two benchmark codes show that such a dynamic approach brings significant benefits over a static state-of-the-art technique.

Keywords. Software Compilation, Data Locality, Memory Layout Optimizations, Array Reuse, Data Dependence.

1. INTRODUCTION

Today’s embedded systems typically involve a mixture of hardware and software components. While the contribution of these components may vary from system to system, we observe an increasing contribution of software. Meeting system-level objectives usually requires flexible software optimizations that can be ported across a wide variety of architectures. The traditional way of writing the embedded software in assembly language does not provide a scalable solution as the system complexity grows. In such cases, high-level languages can replace assembly language as they are cost and time effective, and are easier to maintain. One level of high-level language in programming an embedded system is that the compiler needs to do a very good job if we are to match the quality of the code written in hand-optimized assembly language. To accomplish this, the compiler should be aggressive in optimizing for performance (execution cycles), space, power consumption, and other relevant metrics.

Source-level compiler optimizations (e.g., high-level loop [12, 2] and data transformations [7, 2]) are particularly desirable as they have the potential of achieving large gains. For instance, loop and data layout transformations that target embedded image and video applications can improve iteration-level parallelism and data locality, eventually leading to better utilization of system resources such as functional units and cache memory. In addition, they are also retargetable as they can take as input parameters such as cache capacity, cache block size, and number of functional units.

Optimizing data locality (that is, satisfying majority of data accesses from fast memories such as cache or scratch-pad memory instead of slow main memory) is critical for a large class of embedded codes from image and video processing that manipulate multi-dimensional arrays using multi-level nested loops. To optimize these codes, an optimizing compiler can use either loop-centric optimizations (e.g., those changing the access pattern of nested loops), data-centric optimizations (e.g., those modifying the memory layouts of multi-dimensional arrays), or a combination of these. Most of the previous compiler-based approaches to data locality focus on loop-centric transformations (e.g., loop tiling [12]). While these techniques can be quite successful depending on the application at hand, inherent data dependence in the program being optimized and compiler control structures (e.g., imperfectly-nested loops) may prevent the application of the best loop transformation [5]. Alternatively, data-centric transformations modify memory layouts in an attempt to make them more suitable for the dominant access pattern. These transformations are, however, in general restricted to optimizing spatial locality [7], and demand a more global perspective as an array can be accessed in different places (i.e., not necessarily in a single nest) in a given code using different access patterns. Several research groups (e.g., [8, 9]) have also targeted their research efforts to develop integrated techniques that use loop-centric and data-centric transformations in a unified framework.

Many locality-enhancing approaches that employ data transformations in the domain of regular array codes are limited in their capability in the sense that the layouts determined by these algorithms are static, that is, they are determined at compile time, and valid throughout the entire execution. A disadvantage of this static layout-based locality enhancement strategy is that it fails to optimize codes that manipulate arrays that demand different layouts (from the data locality perspective) in different regions of the code.

This paper presents a novel approach which extends the static layout optimization techniques to select dynamically changing layouts to further improve the locality of data accesses. We refer to this technique as dynamic layout optimization. While previous research on optimizing compilers uses several dynamic layout optimizations (runtime data re-groupings) for irregular applications (e.g.,[4]), to the best of our knowledge, automatic (compiler-based) dynamic layout transformations have not been explored for embedded image and video processing codes with regular (compile-time detectable) access patterns.

Our approach in this paper builds upon our previous work on static layout optimization schemes and employs the techniques developed in [6] as a component (i.e., building block). However, the dynamic optimization approach presented in this paper is flexible, and can work with almost any static locality optimizer that uses some form of layout transformation. Our dynamic layout strategy is also different from the dynamic techniques proposed for irregular applications in an important aspect. In the domain of irregular applications, the layout forms are determined and implemented at runtime depending...
on the interactions between the objects modeled by the application. In contrast, our approach to layout optimization divides the job between compile-time and run-time. More specifically, the layouts of arrays for different regions of the code (program segments) are determined at compile-time (along with the accompanying loop transformations) and the bookkeeping code that is necessary to dynamically transform the layouts of arrays between different program segments is inserted in the code (again at compile time). However, these bookkeeping codes are executed at run-time. In other words, the memory layouts are transformed during the course of execution.

The rest of this paper first revises loop/data transformations and static locality optimizer. After that, our dynamic optimization approach and experimental results are presented.

2. LOOP AND DATA TRANSFORMATIONS

When a reference in a loop nest accesses the same data in different iterations, we say that temporal reuse occurs. Similarly, if a reference accesses data residing on the same cache line in different iterations, we have spatial reuse.

Our focus in this paper is on affine programs in which data structures are restricted to be multi-dimensional arrays, and control structures are limited to sequencing and nested loops. We also allow conditional constructs (e.g., "if statements") between nested loops (but not within them). Loop nest bounds and array subscript functions are affine functions of enclosing loop indices and constant parameters. Under these assumptions, an optimizing compiler can detect the potential temporal and spatial reuse in the code, and convert these reuses into locality (i.e., it can modify the code to exploit these reuses at run-time). Each iteration of the nested loop is represented by an iteration vector, $\mathbf{J}$, which contains the values of the loop index variables at that iteration. Each array reference is an m-dimensional array in a nested loop that contains n loops (i.e., an n-level nested loop) is represented by $\mathbf{L} + \mathbf{d}$, where $\mathbf{L}$ is a vector that contains loop indices. For a specific $\mathbf{I} = \mathbf{J}$, the data (array) element $\mathbf{L} + \mathbf{d}$ is accessed. In this representation, the $mn \times n$ matrix $\mathbf{L}$ is called the access (reference) matrix [12], and the m-dimensional vector $\mathbf{d}$ is called the offset (constant) vector.

The application of a loop transformation represented by a square nonsingular matrix $T$ can be accomplished in two steps [12]: (i) re-writing the loop body, and (ii) re-writing the loop bounds. Assuming that $\mathbf{L}$ is the vector that contains the original loop indices and $\mathbf{L}' = T\mathbf{L}$ is the vector that contains the new (transformed) loop indices, each occurrence of $\mathbf{L}$ in the loop body is replaced by $T^{-1}\mathbf{L}'$. In other words, each reference represented by $\mathbf{L} + \mathbf{d}$ is transformed to $CT^{-1}\mathbf{L}' + \mathbf{d}$. Determination of the new loop bounds, however, is more complicated and may require the use of Fourier–Motzkin elimination.

On the other hand, a data transformation is applied by transforming the dimensions (subscript expressions) of the array reference and the declaration of the associated array [7]. Assuming that we represent the subscript function for a reference as $\mathbf{L} + \mathbf{d}$, a square nonsingular data transformation matrix $M$ transforms this reference to $M\mathbf{L} + \mathbf{d}$. It should be emphasized that applying a data transformation to an array reference (and the array declaration) creates the effect of transforming the memory layout of the array. For instance, a data transformation represented by a square in[ute-unity matrix corresponds to modifying memory layout of a multi-dimensional array from column-major to row-major or vice versa (depending on the language-defined default layout). Assuming that the original reference is $\mathbf{L} + \mathbf{d}$, applying both a loop transformation matrix $T$ and a data transformation matrix $M$ transforms the reference to $MCT^{-1}\mathbf{L}' + \mathbf{d}$. Omitting the offset vector part, since both $\mathbf{M}$ and $\mathbf{T}$ are unknown, finding a suitable $MCT^{-1}$ from the data locality point of view involves solving a non-linear problem, with the constraints such that both $\mathbf{M}$ and $\mathbf{T}$ should be non-singular and $\mathbf{T}$ should observe all the data dependencies in the original nest.

3. DYNAMIC OPTIMIZATION FRAMEWORK

3.1 Static Layout Optimization

The static optimizer employed in this work uses a method that is based on dividing the locality optimization task into loop and data transformations. Let us focus on a single nest. The approach first determines the maximum inherent temporal locality in the nest, and finds a loop transformation (if that is the matrix $T$) to exploit this reuse in the innermost loop. In other words, it specializes the loop transformations to maximize temporal locality. After this scheme is applied, the arrays in the nest are divided into two disjoint groups. The first group contains the arrays with temporal reuse in the innermost loop (as a result of the loop transformation found). The technique does not perform any layout transformations on these arrays (in this nest) as they exhibit temporal locality. The second group is the set of arrays that do not exhibit temporal reuse in the innermost loop (after the loop transformation). Our approach uses data transformations to optimize spatial locality for these arrays. More specifically, for each array, it takes the newly found loop transformation into account and selects the most suitable data layout (i.e., finds a data layout transformation matrix $M$) so as to ensure unit-stride access in the innermost loop. If there are multiple references to the same array, a conflict resolution scheme is used to favor the most dominant (or the most important) reference.

In order to handle multiple nests inside a program-wide optimization, the technique in [6] propagates memory layouts across loop nests. To accomplish this, it first orders the nest using an importance criterion (also called cost criterion). The first (the most important) nest is then optimized using the strategy explained in the previous paragraph. After this, the memory layouts for a subset (possibly all) of the arrays in the nest are determined. The rest of the nest are determined next. The algorithm moves to the next most important nest. It uses the same approach (as in the most important nest) to optimize its memory layout. Each array reference that, in selecting the most suitable loop transformation, it also takes the memory layouts that have been found so far into account. Again, a conflict resolution scheme is used to achieve both the objectives. After optimizing this nest, a group of new layouts is added to the 'set of determined layouts', and the approach moves to the next (third) most important nest and optimizes this nest, taking into account all the layouts found so far (from the most important nest and the second most important nest), and so on. A complete discussion of the static optimizer can be found in [6].

3.2 Program Representation

In our framework, each procedure is represented using a nest flow graph (NFG) which can be described as follows. Each node $v$ in the NFG represents a nested loop in the code and each directed edge (arrow) from $v$ to $v'$ indicates that there might be a flow of control from the nest represented by $v$ to the nest represented by $v'$. Each edge $e = (v, v')$ also carries a weight which is the product of the estimated frequency of control transfers (transitions) from node $v$ to node $v'$ and the number of the exposed misses in $v'$ (denoted $m_v$).

The exposed misses in $v'$ are the misses incurred assuming some fixed memory layouts for the arrays accessed in the nest represented by $v'$. Note that in our framework different arrays may have different memory layouts.

As an example, Figure 1 gives a code sketch and its NFG. In a sense, we can think of an NFG as a simplified macro control flow graph (CFG) where each node corresponds to a nested loop instead of basic block. Let us now try to understand how $f_{v,v'}$ (the estimated number of control transitions from $v_1$ to $v_3$) is determined. This depends on two factors: the probability with which the conditional branch at the end of $v_1$ will take the route to $v_2$ and the number of times the outer loop iterates. Supposing that the branch (at the end of $v_1$) is taken 50% of the time and the said loop iterates $N$ times, then $f_{v_1,v_2}$ is $0.5N$. Similarly, $f_{v_3,v_4}$ is 0.5N, assuming that the loop that encloses only the nodes $v_2$ and $v_4$ iterates $N'$ times. Note that these frequencies do not consider the number of times a given nested loop (represented by a node in the NFG) is executed.

220
They just consider the execution frequencies of the edges between the nested loops (nodes). It should be noted that while a given $f_{uv}$ is an important factor in determining the weight (or importance) of the edge $(u, v)$, it is not sufficient alone. This is because if the number of exposed misses in $v$ is very low, the control flow on $(u, v)$ is not a major determinant factor in shaping the locality behavior of the code. Consequently, we also need to get $m_{uv}$ (the estimated number of exposed misses) in the picture. Overall, we can define $w_{uv}$, the weight of the edge between $u$ and $v$, as $m_{uv} f_{uv}$.

### 3.3 Estimating the Number of Misses

In this subsection, we discuss how to estimate the number of exposed misses, $m_{uv}$, for a given nested loop represented by $v$. Our approach is modeled after the technique proposed by McKinley et al. [8]. An important modification that we made to the original miss estimation algorithm is to take into account the possibility that the different arrays referenced in the nest can have different memory layouts. The approach in [8] first groups the references according to the potential group-reuse between them. Then, for each representative reference, it calculates a reference cost (i.e., the estimated number of misses during a complete execution of the innermost loop).

Basically, the reference cost of a given array reference wise, respect to a loop order is 1 if the reference has temporal reuse in the innermost loop; that is, the subscript functions of the reference are independent of the innermost loop index. The reference cost is $irvp/(cl/cstride)$ if the reference has spatial reuse in the innermost loop. In this expression, $irvp$ is the number of iterations of the innermost loop (trip count), $cl$ is the cache line size in data items, and $cstride$ is the stride size of the innermost loop multiplied by the coefficient of the loop index variable. Finally, if the reference in question exhibits neither temporal nor spatial reuse in the innermost loop, its reference cost is assumed to be equal to $irvp$; that is, a cache miss per iteration. In this latter case, the overall loop cost of the nest is the sum of the contributions of each reference it contains. The contribution of a reference is its reference cost multiplied by the number of iterations of all the loops (that enclose the reference) except the innermost one. Note that this miss calculation process is a good first degree approximation if one does not consider conflict misses and internal data reuse.

In our framework, we allow different arrays to have different memory layouts; consequently, we need to relax the fixed uniform layout requirement. Consider the following two-level nested loop that accesses three two-dimensional arrays:

```plaintext
for $i = 1, N$
for $j = 1, N'$
$U[i][j] = V[i][j] + W[i][j]$
end for
end for
```

Assuming that arrays $U$ and $V$ have column-major layouts and array $W$ has a row-major memory layout, it is easy to see that the reference cost of array $U$ is $N'/cl$, the reference cost of array $V$ is $N'/cl$, and the reference cost of array $W$ is $N'$. In fact, it is possible to modify the original algorithm in [8] as follows to accommodate different layout forms. Assuming that the language-defined default memory layout is column-major, the impact of a row-major layout can be imitated by pre-multiplying the reference matrix with the inverse of the identity matrix (which corresponds to row-major to column-major layout conversion). In our example nest above, this process transforms the reference to array $W$ from $W[i][j]$ to $W[j][i]$. After this, the miss estimation process developed in [8] (which assumes column-major layout for every array) can be applied to $W$ as well. Continuing with the example, the contributions of the arrays $U$, $V$, and $W$ are $N'/cl$, $N'/cl$, and $N'$, respectively. Therefore, the overall cost (the estimated number of exposed misses) is $N'(1+2/cl)$. In the remainder of this paper, when we mention cache miss, we mean exposed cache misses.

### 3.4 Algorithm

The most important part of our approach is determining the layouts of arrays at different program points (i.e., in different nests). Our technique makes use of the static optimizer and the cache miss estimation technique explained above. Let us first focus on a simple case where we have a number of consecutive loop nests with no conditional statements between them and no timing/convergence test loops. As an example of such a code and its NFG are shown in Figures 5(a) and (b), respectively. Our dynamic optimization approach proceeds as follows. First, it uses the static locality optimizer to optimize nest $v_0$ through nest $v_4$ (i.e., the entire code). This optimization returns a potentially modified code with optimized nests (through loop transformations) and assigns a suitable memory layout to each array. This is normally the code that would be returned by the static approach described in [6]. The dynamic approach estimates the number of misses for this optimized code and records it. Instead of modifying the original code directly (as would be done by the static optimizer), it just records the loop and data transformations found along with the number of estimated misses (call this number $cost_{42}$).

Next, it checks all the edge weights and selects the edge with the highest weight as the cut point. Without loss of generality, let us assume that in our example the cut point is $(v_4, v_5)$, that is, the edge between $v_4$ and $v_5$. This cut point divides the code into two logical partitions: the one that contains the nests $v_0, v_1$, and $v_2$ and the other one that contains $v_3$ and $v_4$ (see Figure 3(c)). After that, the dynamic approach runs the static optimization algorithm for each partition separately, and obtains the number of misses for each of them (call these numbers $cost_{43}$ and $cost_{34}$). It also records the preferable layouts and accompanying loop transformations for each partition. Note that since these two partitions may access some common arrays, it is possible that the static optimizer can select different layouts for the same array in each partition [5]. If there are such arrays, the dynamic approach also calculates $overhead_{34}$, the estimated number of misses that would occur during dynamically transforming the layouts of such arrays between two partitions. The approach then compares $cost_{43}$ and $cost_{42} + cost_{34} + overhead_{34}$. If the former term is smaller than or equal to the latter, the approach stops and returns the result of the static optimizer (when the input is the entire code) as output (indicating that the static layout optimized version is the best alternative for this code). If not, this means that transforming memory layouts across partitions (dynamic layout transformation) might be beneficial for the code in question (under our miss estimation model). In this case, the algorithm recursively applies this strategy by dividing the two partitions in Figure 3(c) into further subpartitions. At each step of the recursion, the algorithm re-computes the total number of misses (considering the entire code) and compares this figure with the case before the recursion and stops the recursion if and only if further partitioning the code could not improve the results.

The notation $cost_{ij}$ refers to the number of estimated misses for the program fragment starting with $v_i$ and ending with $v_j$ (including both $v_i$ and $v_j$). On the other hand, $overhead_{ij}$ denotes the number of estimated misses that would occur during dynamically transforming the memory layouts (whenever necessary) between the partitions that ends with $v_i$ and the partition that starts with $v_j$.
INPUTS: A procedure $P$ that accesses a number of array elements
OUTPUTS: Optimised code with transformed nested loops and
memory access locality.

Begin

Build a Data Flow Graph (DFG) for the procedure
\[ P^* = \text{static-optimizer}(P) \]
\[ \text{cost} = \text{cost-calculator}(P^*) \]
Compute an edge weight $w_{uv}$ for each edge $(u, v)$ in $P^*$ as $w_{uv} = f_{uv} \cdot m_{uv}$.
\[ \text{current-min} = \text{cost} \]

Build an acyclic NFG (ANFG) from the DFG by:
- eliminating the loop edges, and
- adjusting edge weights as $w'_{uv} = f_{uv} \cdot w_{uv}$ where $P'$ is the combined frequency of all loops in the NFG that access $(u, v)$.

Determine the levels in the NFG.
Select a cut point and determine two logical partitions $P_1$ and $P_2$ from $P$.
\[ P_1 = \text{static-optimizer}(P_1) \]
\[ P_2 = \text{static-optimizer}(P_2) \]
\[ \text{current-min} < \text{cost-calculator}(P_1) + \text{cost-calculator}(P_2) + \text{cost}(P_1, P_2) \]
Return $(P')$
\[ \text{current-min} = \text{cost-calculator}(P_1) + \text{cost-calculator}(P_2) + \text{cost}(P_1, P_2) \]
\[ \text{cost-calculator}(P) \]
\[ \text{cost-calculator}(P) \]
End

Procedure $\text{cost-calculator}(P')$

Begin

Determine levels in $P'$. 
Select a cut point and determine two logical partitions $P_1$ and $P_2$.
\[ P_1 = \text{static-optimizer}(P_1) \]
\[ P_2 = \text{static-optimizer}(P_2) \]
\[ \text{current-min} < \text{cost-calculator}(P_1) + \text{cost-calculator}(P_2) + \text{cost}(P_1, P_2) \]
\[ \text{current-min} = \text{cost-calculator}(P_1) + \text{cost-calculator}(P_2) + \text{cost}(P_1, P_2) \]
\[ \text{cost-calculator}(P) \]
\[ \text{cost-calculator}(P) \]
End

Figure 2: Dynamic locality optimisation algorithm.

Figure 3: (a) An example code sketch; (b)-(f) Different logical partitions.

results in a higher cost than the current minimum or there is no possibility of partitioning the code further (recall that our approach works on a nest granularity). During the optimization process, it also keeps track of the current minimum.

Returning to our example, supposing that after the first partitioning step, \( \text{cost}_{02} + \text{cost}_{24} + \text{overhead}_{23} > \text{cost}_{04} \), the algorithm records \( \text{cost}_{02} + \text{cost}_{24} + \text{overhead}_{23} \) as the current minimum and further partitions the code. Assuming that in the first partition \((v_0, v_2)\) is the cut point, then the dynamic optimization approach applies the static optimizer to two subpartitions: one containing the nests \( v_0 \) and \( v_1 \) and the other containing only the next \( v_2 \) (see Figure 3(d)). The new total (procedure-wide) cost is computed as \( \text{cost}_{24} + \text{cost}_{02} + \text{overhead}_{23} \) and is compared to the current minimum. Note that, in general, \( \text{overhead}_{23} \) might be different from \( \text{overhead}_{23} \) as the static optimizer can return different layouts for the same nest partition, depending on whether it is working on \( v_0, v_1, \) and \( v_2 \) (as a single partition) or only on \( v_2 \). Continuing with the example, assuming that \( \text{cost}_{02} + \text{cost}_{24} + \text{overhead}_{23} \) is smaller than the current minimum, the dynamic approach selects this as the current minimum and this time it checks the profitability of splitting the partition that contains the nests represented by \( v_0, v_1, \) and \( v_2 \). Assuming that this partitioning and any further partitioning of the subpartition that contains the nests \( v_0, v_1, \) and \( v_2 \) do not bring any benefit, the approach stops and returns the three partitions (their optimised versions) as shown in Figure 3(d). Figures 3(e) and (f), on the other hand, show what would be the case if further partitionings were beneficial.

So far, we have assumed that there is no control flow between loop nests. In the presence of a control flow, determining the cut points and estimating the number of misses (especially the ones due to explicit layout transformation overhead, that is, due to transforming layouts dynamically between loop nests) are more difficult. Let us consider once more the code sketch and its NFG in Figure 1. In the first step of our dynamic approach, the static optimizer attempts to optimize the entire code (that includes all the nested loops \( v_0 \) through \( v_2 \)) without taking into account the control flow explicitly. After analyzing the code and recording the loop and data transformations and the estimated number of misses (for the optimized version code), the approach selects a cut point. Due to the structure of the program, any cut now may need to cross multiple edges in the NFG. For instance, assuming that \((v_0, v_2)\) (in the NFG shown in Figure 1) is the cut point, Figure 4(a-d) show four alternatives for dividing the program into two logical partitions. It is not clear which alternative is the best choice.

Our approach first eliminates the back edges in the NFG and transforms the NFG to an acyclic graph (called acyclic NFG or ANFG) in which the edge weights are the original weights (as defined above) multiplied by the frequencies of all enclosing back edges. For example, in Figure 4(e), \( w_{24} \), the weight of the edge between the nodes 3 and 4 in the ANFG is computed as \( w_{24} = w_{24} \cdot f_{24} \cdot f_{23} \); as both \((v_0, v_2)\) and \((v_0, v_1)\) enclose \((v_2, v_4)\) in the original NFG (Figure 1). Similarly, it is easy to see that \( w_{12} = w_{12} \cdot f_{24} \cdot f_{23} \). The ANFG for this example is shown in Figure 4(e). Afterwards, our approach divides this ANFG into levels as illustrated in Figure 4(f). Each level corresponds to a potential cut point. It then calculates a level weight for each level which is the sum of the weights of all the edges that intersect the level. Finally, it selects the level with the largest level weight as the cut point and divides the ANFG into two partitions. It then applies the static optimizer to both of the partitions. For our example in Figure 4(e), it compares \( w_{12} + w_{13} + w_{24} + w_{34}, \) and \( w_{12} < w_{13} \) and assumes that \( w_{12} + w_{13} \) is greater than the others, it selects as the cut point the level that intersects with the edges \((v_0, v_3)\) and \((v_0, v_3)\); that is the second level from top in Figure 4(f). This gives us two partitions: one that contains the nests \( v_0, v_1, \) and \( v_2 \) and the other containing the nests \( v_2, v_3, v_4, \) and \( v_5 \). It then recursively applies the static optimizer to these partitions. It should be noted that computing the overhead costs in this case can be done in the same way as in the previous case (without control flow). A sketch of our dynamic optimization algorithm is given in Figure 2. In this algorithm, static-optimizer is a routine that takes as input a logical partition and returns as output its layout-optimized version. cost-calculator estimates the total number of
of misses, and current-ixa holds the current minimum (the best total number of misses found so far). The main program computes the total number of misses, and then (if beneficial) divides the program into two logical partitions, and calls the compute routine for each partition.

3.5 Transforming Memory Layouts between Nests

Once optimal layouts in different program parts and accompanying transformed loops are determined, the next step is modifying the code. Transforming array layouts (using data transformations) within loop nests has already been discussed by previous research [3, 7, 6]. However, transforming memory layouts explicitly between loop nests has not been previously addressed. In the following, we address this issue by dividing the problem into two components: (i) determining optimal points in the code to insert layout transformation (copy) loops, and (ii) structuring the copy loops to minimize loop overhead.

Explicitly transforming memory layout of a given array is typically done through the use of a copy loop. For example, to convert layout of a two-dimensional array $U$ of size $N \times M$ from row-major to column-major, we can use a two-level nested loop using which the transpose of $U$ is copied to another array $U'$. After the execution of the copy loop, the array identifier $U'$ is used instead of $U$.

Note that the copy loops used to implement explicit layout transformation are pure overhead, as they should be optimized as much as possible. This can be achieved by selecting the most appropriate points to transform arrays, and by transforming the layout of multiple arrays simultaneously. Let us first focus on an NFG without a conditional control flow between loop nests. Figure 5(a) shows such an NFG. We assume that this NFG uses five different arrays. For the sake of simplicity, we also assume that each array needs to be transformed just once. It is easy to extend our strategy to cases where the array layouts need to be transformed multiple times. For each array, we define a slack as the set of consecutive edges, along which the array in question can be layout transformed. Figure 5(a) shows five illustrative slacks corresponding to five arrays, one of the arrays, one of the slacks contains three edges: $(v_1, v_2, v_3)$, and $(v_3, v_4)$. Our strategy is based on the assumption that minimizing the number of actual layout transformation points is critical. This assumption is reasonable as minimizing the number of transformation points might enable us to transform a number of arrays simultaneously (i.e., using the same copy loop). In the example shown in Figure 5(a), this can be done if we select the points marked "**" as the layout transformation points. Note that in the first transformation point (between $v_2$ and $v_3$), we can potentially transform three arrays together, and in the second transformation point (between $v_3$ and $v_4$), we can transform (again potentially) two arrays. Whether or not these simultaneous layout transformations can actually be done depends on a large extent to the dimensionalities and sizes of the arrays. On the other hand, in this example, a scheme that does not aim at minimizing the number of layout transformation points can use five different transformation points in the worst case (marked using "*" in Figure 5(a)).

When we take conditional control flow into account, the problem of selecting transformation points becomes more difficult. Consider the NFG in Figure 5(b) assuming that $(v_1, v_2)$ is the only edge in the slack for a given array and $v_1$ and $v_2$ are the only using that array. Obviously, this means that the array in question should be transformed in the program portion corresponding to this edge. However, the exact point along this edge at which the array is transformed can make a great difference in performance. For example, the transformation can be done before the "if statement" that follows $v_1$ (marked "+" in Figure 5(b)). However, during execution, if the branch that goes into $v_2$ is taken, this transformation would be useless. A better option is transforming the array after the 'else portion' of the 'if statement' (see the code sketch in Figure 1). This point is marked using "*" in the edge $(v_1, v_2)$ in Figure 5(b). In this case, the layout transformation overhead will be incurred only if it is really necessary. Based on this observation, our current implementation postpones the layout transformations as much as possible. Another issue here is determining the point to transform the layout back to the original form. In Figure 5(b), this transformation should be done along the edge $(v_3, v_4)$.

Now assume that for a given array the desired layout by the nest represented by $v_3$ (see Figure 5(e)) is different from the other nests. One alternative in this case is transforming the layout at the two points marked "**" to satisfy the layout requirement imposed by $v_3$. In Figure 1, this corresponds to the points between nest $v_3$ and the 'else statement' and between the second 'if statement' and nest $v_3$. A better alternative, however, is transforming the array only once just before $v_3$ (marked "*"). As in the previous case, the array should be transformed back along the edge $(v_3, v_1)$. The remaining two NFGs in Figure 5 illustrate two more transformation cases. In Figure 5(c), we assume that for a given array the desired layout by the nest represented by $v_1$ is different from the other nests. In this case, the array needs not be transformed back as $v_1$ is executed only once (i.e., there is no back edge). Finally, in Figure 5(d), we assume that only the nests $v_3$ and $v_4$ demand a different layout from others for a particular array. The transformation points are marked using "*".

Now focus on how multiple arrays can be layout-transformed using the same copy loop. Suppose for example that two two-dimensional arrays $U$ and $V$ of sizes $N \times N$ and $N' \times N'$, respectively (where $N' \geq N$), are to be layout-transformed at the same point in the code. To handle this, a naive approach would use two different copy loops as shown in Figure 6(a). Assuming a loop iteration overhead of $C$ (per iteration), the total cost of these loops is roughly $N^2 C^* + N^2 C'$ instead. We could implement these transformations as shown in Figure 6(b). Under the assumption mentioned above, the total loop cost of these copies now is $N^2 C^* + (N' - N) N C'$ which is equivalent to $2N^2 C + N^2 C' - 2N N' C$. Since $N' \geq 2N$, this second alternative is preferable over the first one. Note that this optimization can be generalized to multiple arrays and non-square arrays with minor modifications.
Figure 7: Performance results for two benchmarks. ‘M’ means million and ‘B’ means billion.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results for two codes: a two-dimensional Fourier Transform (FT) and a two-dimensional Alternating Direction Implicit Method (ADI). The total input sizes are 15MB and 16.5MB for FT and ADI, respectively. We implemented our approach using the Parafrase-2 compiler [10]. Since our approach uses the static optimizer as a subcomponent, the additional code to implement the dynamic partitioning strategy was less than 700 lines.

The experiments are run on a MIPS R10K-based machine. The R10K is a microprocessor that can fetch and decode four instructions per cycle and can run them on five pipelined functional units. The processor implements the MIPS IV instruction set architecture and has a two-level cache hierarchy. Located on the chip are a 32KB, two-way set associative Level-1 (L1) instruction cache and a 32 KB, two-way set associative, two-way interleaved Level-1 (L1) data cache. Off-chip is a two-way set associative, unified Level-2 (L2) cache (4MB). For the L1 cache hits, the latency is two to three cycles; and for L1 misses that hit in the L2 cache, the latency is eight to ten cycles. The main memory latency is at least sixty cycles.

Figure 7 gives the number of L1 (data) and L2 misses, MFLOP rates, and execution cycles for four different versions of the two benchmarks. The original version represents the unoptimized code, and static represents the version optimized using only the static optimizer [6]. The dynamic version is the result of the strategy discussed in this paper (our approach). Finally, sdp gives the optimal solution based on integer linear programming (ILP). This last version is not implemented in the compiler, but obtained by formulating the ILP problem for dynamic layout optimization (assuming only row-major and column-major layouts) and solving it using the lp.solve package [11]. We omit the details of the ILP formulation due to lack of space. For static and dynamic, we fed the input code (in C) to Parafrase-2, and the optimized code returned by Parafrase-2 (again in C) has been compiled by the native compiler for R10K using the O2 optimization flag.

These results show that, in FT, the dynamic approach resulted in the optimal solution returned by ILP. Our approach reduces the number of cycles by 50.9% as compared to the static optimization. In ADI, on the other hand, there is a difference between sdp and dynamic as far as the performance is concerned. We believe that this is due to our miss estimation model rather than the layout selection strategy. Nevertheless, as compared to the static version, we achieve a 65.5% improvement in execution cycles. These results indicate that the dynamic optimization strategy is very successful.

5. CONCLUSIONS AND ONGOING WORK

This paper introduces a dynamic layout optimization strategy to minimize the number of cycles spent in memory access. A given multi-dimensional array is allowed to have different memory layouts in different parts of the application if doing so improves data locality beyond the static approaches that fix memory layouts to specific forms at compile-time. In this dynamic strategy, different layouts that a given array will assume at run-time are determined at compile-time; however, the layout modifications, themselves, occur dynamically during the course of execution. Currently, we are experimenting with different static optimizers and cache miss estimation techniques within the dynamic optimizer. We plan to extend our approach to employ more than one static optimization strategy, and activate the most suitable one at runtime.

6. REFERENCES


