Interprocedural Optimizations for Improving Data Cache Performance of Array-Intensive Embedded Applications

W. Zhang, G. Chen, M. Kandemir
CSE Department
The Pennsylvania State University
University Park, PA 16802, USA
{wzhang,gchen,kandemir}cse.psu.edu

M. Karakoy
Department of Computing,
Imperial College,
London, SW7 2AZ, UK
m.karakoy@ic.ac.uk

ABSTRACT

As datasets processed by embedded processors increase in size and complexity, the management of higher levels of memory hierarchy (e.g., caches) is becoming an important issue. A major limitation of most of the cache locality optimization techniques proposed by previous research is that they handle a single procedure at a time. This prevents compilers from capturing the data access interactions between procedures and may result in poor performance. In this paper, we look at loop and data transformations from a different angle and use them in an interprocedural optimization framework. Employing the call graph representation of a given application, the proposed technique visits each node of this graph twice and uses loop and data transformations in a systematic way for optimizing array layouts whole program wide. Our experimental results show that this interprocedural locality optimization strategy is much more effective than the previous locality-based techniques that handle each procedure in isolation.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Design Styles—*Cache memories*; D.m [**Software**]: MISCELLANEOUS

General Terms

Algorithms, Performance

Keywords

Cache, Locality, Embedded System

1. INTRODUCTION

Many previously-proposed compiler-based techniques to data locality focus on a single procedure at a time. While this makes implementation of these techniques simple to manage, it also entails a performance penalty when arrays are shared between procedures. Since these *intraprocedural* techniques optimize each procedure in isolation, data sharing between procedures cannot be captured. As a result, important optimization opportunities might be missed. While it is possible to partly capture the impact of interprocedural data sharings by making conservative assumptions across procedure boundaries, in most cases such assumptions are not accurate enough to handle powerful optimizations (especially, those targeting loop-level parallelism and data locality).

To demonstrate how an interprocedural optimization can help to improve performance, we consider the program fragment shown in Figure 1. In this fragment, the main procedure (`main()`) updates an array and then passes it to a subroutine named `Proc()`. Suppose that we want to determine the most suitable memory layout for array `X`. If one considers only the main procedure, a row-major memory layout seems to be the best alternative (as the array is accessed in a row-by-row fashion). However, if we also consider the access pattern in `Proc()`, we can see that a column-major memory layout is better alternative as there are four column-by-column accesses to the formal parameter `Y` (which corresponds to the actual parameter `X`). This small example shows that interprocedural analysis can help the compiler select better optimization strategy (in this case, a better memory layout) than a pure intraprocedural analysis. Note that if an (alternative) approach analyzes both the main procedure and `Proc()` separately (i.e., without seeing the other procedure), it can select a column-major layout in `Proc()` and a row-major layout in the main procedure. Consequently, a (dynamic) layout transformation might be required between these two procedures. An interprocedural optimization strategy can, on the other hand, see the big picture and select the most suitable (program-wide) memory layout without resorting to dynamic layout transformation.

In optimizing data cache behavior of array-intensive applications, two different techniques have emerged. The first technique, called loop transformations, focus on loop nests and restructures them for data locality (e.g., [10, 8, 2]). The second technique, called data transformations, focus on memory space (instead of iteration space) and determines the most suitable memory layouts for multi-dimensional arrays (e.g., [7, 2]). There also exist integrated techniques (e.g., [9, 5]) that combine loop and data transformations in a unified setting. However, *none of these previous studies has focused on inter-procedural locality optimization problem.* In this paper, we look at loop and data transformations from a different angle and use them in an *interprocedural optimization framework.* Employing the call graph representation [11] of a given application, the proposed technique visits each node of this graph twice and uses loop and data transformations in a systematic way for optimizing array layouts whole program wide. Our application domain is array intensive benchmarks that are used frequently in embedded image/video applications.

To test the effectiveness of our approach, we implemented it in a source-to-source translator and applied it to six array-intensive

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

```
main()
 {
   for i = 1, N
     for j = 1, N
       X[i][j] = ...
   ...
   call Proc(X,...)
   ...
 }

 Proc(Y,...)
 {
   ...
   for j = 2, N-1
     for i = 2, N-1
       k += (Y[i][j-1]+Y[i][j+1]
            +Y[i-1][j]+Y[i+1][j])/4.0
   ...
 }
```

Figure 1: A program fragment that can benefit from interprocedural locality optimization.

embedded applications. The experimental results presented in this paper indicate that our interprocedural locality optimization strategy improves the performance of array-intensive applications significantly. The results also show that the interprocedural optimization is much more effective than the previous locality-based techniques that handle each procedure in isolation. Based on these results, we strongly encourage compiler writers for embedded systems to incorporate interprocedural transformations to their suite of optimizations. While a previous study [6] has also considered inter-procedural transformations for optimizing locality, there are several important differences between our approach and that work. First, the problem formulations are different. The mentioned paper adopts a graph-based approach based on "maximum-branching" (even working within a procedure), whereas our approach does not use a graph representation. Second, the approach used in [6] is entirely static; while the approach presented in this work can make use of a dynamic strategy as well (where the data layouts are transformed between the procedures during the course of execution). Third, these two techniques have been evaluated in different domains. The work in [6] targets scientific codes while our approach focuses on embedded applications. While the inter-procedural code analysis and optimization problem, itself, is not novel (e.g., see [11] and the references therein), its application to cache locality optimization of array-intensive applications has not taken much attention from the previous research.

The rest of this paper is organized as follows. Section 2 revises the call graph representation and gives a summary of our overall optimization strategy. It also gives background on representation of affine programs. Section 3 explains how locality constraints are computed. Section 4 presents the bottom-up phase of our call graph analysis. Section 5 explains how program-wide locality constraints are solved. Section 6 presents the top-down phase, focusing in particular on how loop transformations can be exploited for satisfying the inherited memory layouts. Section 7 discusses how our approach can be extended to accommodate dynamic layout transformations. Section 8 gives details of our implementation and presents experimental data. Finally, Section 9 summarizes our major conclusions.

2. PROGRAM REPRESENTATION AND OVERALL STRATEGY

A call graph is a directed graph, $G = (V, E)$. The finite set of nodes, V, consists of the procedures that may be called in the program. For any two procedures (nodes) f and g in V if there is a potential call to g by f then the (directed) edge (f, g) appears on the graph. The complete collection of directed edges is denoted by E. The root node of a call graph $G = (V, E)$ is a procedure which is not called by any other procedure. This node corresponds to the main procedure in a given C program. While, if desired, the edges of the call graph can be marked using information about parameters passed/results returned, in this study, it is sufficient to work with a plane (unmarked) call graph.

Our overall approach to interprocedural analysis for data locality works as follows. First, we build locality constraints for each procedure in the application. These locality constraints capture a set of equations such that solving these equations gives us suitable memory layouts for the arrays referenced in the procedure. However, while an intraprocedural optimization strategy would solve these constraints and determine memory layouts, our interprocedural strategy propagates these constraints to the parent procedures in the call graph. In this way, the constraints are propagated up in the call graph until they reach the root node (which represents the main procedure in the application). This constraint propagation step is referred to as the bottom-up phase in the rest of this paper. The locality constraints are then solved at the root and the arrays are assigned memory layouts. After this step, a top-down phase starts and the memory layouts found are propagated down to the children nodes. In receiving the memory layouts from its parents, a node (procedure) is optimized using loop transformations. The following sections give details of this interprocedural strategy and presents experimental data.

We say a temporal reuse exists when two array references access the same data element. Spatial reuse, on the other hand, occurs when two references access the same transfer unit such as a cache line [11]. When a loop nest exploits temporal (spatial) reuse, we say that the associated references exhibit temporal (spatial) locality. It should be noted that reuse does *not* necessarily mean locality [8]. To illustrate the concepts of reuse and locality, we consider the following example.

```
for i = 1, N
  for j = 1, N
    X[i] = Y[j] + Z[i][j] + T[j][i]
```

Assuming a column-major layout as the default, in this loop nest, array `X` has temporal reuse in the `j` loop, and spatial reuse in the `i` loop. Array `Y` has temporal reuse in the `i` loop, and spatial reuse in the `j` loop. Arrays `Z` and `T`, on the other hand, have only spatial reuses in the `i` and `j` loops, respectively; they do not exhibit any temporal reuse as each element of `Z` and `T` is accessed only once (during the execution of the nest). Assuming that `N` is very large, only the reuses associated with the `j` loop exhibit locality [10]. As a result, the exploitable reuses (that is, the reuses that can be converted into locality) for this nest are the temporal reuse for `X`, and the spatial reuses for `Y` and `T`.

Note that if array `Z` is stored in row-major order instead of the default column-major order, then spatial reuse for it can also be exploited in the innermost loop. In this paper, we address the problem of selecting appropriate memory layouts for multi-dimensional arrays as well as suitable iteration space transformations in a global (application-wide) manner. To do this, we employ an interprocedural analysis framework.

The optimization strategy presented in this paper relies on results from the loop transformation theory [8, 11, 10]. Specifically, we focus on loops where both array subscripts and loop bounds are affine functions of enclosing loop indices and loop-invariant constants. A reference to an array `X` is represented by $R_x\vec{I} + \vec{r_x}$, where R_x is a linear transformation matrix called the array reference (access) matrix, $\vec{r_x}$ is the offset (constant) vector [10]; $\vec{I}$ is a column vector, called the iteration vector, whose elements written left to right rep-

resent the loop indices $i_1, i_2, \cdots, i_n$, starting from the outermost loop to the innermost loop in the nest. For the rest of the paper, a reference to array X will be denoted by a pair $(R_x, \vec{r_x})$. To clarify these concepts, we consider the following loop nest.

```
for i = 1, N
  for j = 2, N/2
    for k = 1, N/2
      X[i][j] = Y[j+k][i][j-1] + 2
```

The reference matrices and offset vectors are as follows:

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \vec{r_x} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

and

$$R_y = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \vec{r_y} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}.$$

Linear mappings between iteration spaces of loop nests can be modeled by square, non-singular (loop transformation) matrices [8, 11]. If $\vec{I}$ is the original iteration vector, after applying linear transformation (represented by matrix T), the new iteration vector is $\vec{I'} = T\vec{I}$. Similarly if $\vec{d}$ is the distance (resp. direction) vector, on applying T, $T\vec{d}$ is the new distance (resp. direction) vector [11]. Since $\vec{I} = T^{-1}\vec{I'}$, after the transformation T, R_xT^{-1} is the new (transformed) array reference matrix [11].

Similarly, a data transformation for an array X is expressed using a data transformation matrix M_x. Applying a data transformation transforms a reference $(R_x, \vec{r_x})$ to $(M_xR_x, M_x\vec{r_x})$. It also modifies the corresponding array declaration. More details on data transformations and low-level code generation issues can be found elsewhere [7]. In this work, we assume that the memory layout for an h-dimensional array can be in one of $h!$ forms, each corresponding to the linear layout of data in memory by a nested traversal of the (dimension) axes in some predetermined order. The innermost axis is called the *fastest-changing dimension*. As an example, for a row-major memory layout of a two-dimensional array, the second dimension is the fastest changing dimension. Since in many array-intensive embedded computations, the arrays are very large, it might be difficult to exploit data locality beyond the fastest changing dimension. Even though the order of other dimensions may impact the performance, their effect is of secondary importance.

3. LOCALITY CONSTRAINT GENERATION

In this section, we discuss how we generate locality constraints for each procedure in the program. Each local constraint has two components: a *weight* and an *equation*. The weight indicates how important to satisfy the corresponding equation. The equation component, on the other hand, is obtained using the data access pattern (in the nested loop) and the data transformation required. Let $(R_x, \vec{r_x})$ be a reference to an array X in a given nest whose iteration vector is $\vec{I}$. In order to improve the cache behavior of accesses through this reference, we need to select a data transformation matrix M_x such that the last column of $M_xR_x\vec{I} + M_x\vec{r_x}$ should contain the index of the innermost loop with a small coefficient and this loop index should not appear in any other column. In other words, the following constraint should be satisfied:

$$M_xR_x = \begin{pmatrix} u & u & u & \ldots & u & 0 \\ u & u & u & \ldots & u & 0 \\ u & u & u & \ldots & u & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ u & u & u & \ldots & u & c \end{pmatrix}, \qquad (1)$$

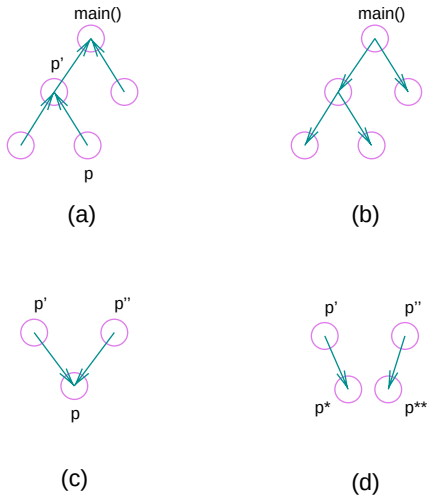

Figure 2: (a) Bottom-up phase. (b) Top-down phase. (c) A call graph fragment that contains a node with multiple parents. (d) Applying procedure cloning to the fragment in (c).

where u denotes a *don't care* entry and c is a small value (preferably 1).[1] Note that it may or may not be possible to find such an M_x matrix. Note also that such a constraint can be written for each array reference occurring in each nest in the procedure. Each such constraint is referred to as the *locality constraint* as it imposes a condition that needs to be satisfied by the data transformation matrix M_x if we are to optimize the locality behavior of the corresponding reference.

Let $LC_{i,j,k,p}$ denote the locality constraint due to the kth reference to array X_j in nest i of procedure p. The set of locality constraints of procedure p is denoted by

$$LC(p) = \bigcup_i \bigcup_j \bigcup_k LC_{i,j,k,p}.$$

That is, it contains all constraints due to all arrays, all references, and all loop nests.

4. BOTTOM-UP PHASE: CONSTRAINT PROPAGATION

In this section, we explain how locality constraints are propagated up in the call graph. Let p be a procedure and p' be its parent. In processing the procedure p (i.e., in optimizing it), we build $LC(p)$ and propagate a *subset* of it to p'. This subset excludes the constraints $LC_{i,j,k,p}$ such that X_j is a local array of p. There is no need to propagate these constraints as the corresponding layouts (i.e., the layouts of the local arrays) are important only for p. In this way, for each node in the call graph, a subset of the local constraints built for the corresponding procedure is propagated up to its parent(s). This process is depicted in Figure 2(a) for an example call graph that contains five procedures. At the end of this process, at the root (that is, the main procedure), we have all constraints that involve non-local arrays coming from all procedures in the application. We add to this set all locality constraints of the root itself,

[1] It is known from data transformation theory [7] that if a reference originally does not exhibit temporal reuse, it is not possible to transform it to obtain temporal reuse using data transformations alone. Therefore, if the original matrix entry in place of c (that is, the last row-last column element) is not 0 (i.e., it does not exhibit temporal reuse), we cannot select a M_x such that c becomes 0.

and obtain a (potentially) large set of constraints.

An important issue in propagating constraints is to re-map the locality constraints in procedure p to constraints that can be used in conjunction with the constraints in p' (the parent of p). For example, suppose that the procedure p has two references, $Y[i][j]$ and $Z[j][i]$, in a nest with loops i (the outer) and j (the inner). So, the locality constraints for this procedure (which has only two formal parameters: Y and Z) are:

$$M_y \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} u & 0 \\ u & 1 \end{pmatrix} \quad \text{and} \quad M_z \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} u & 0 \\ u & 1 \end{pmatrix}.$$

Note that if these constraints are considered as they are, it is easy to find suitable data transformation matrices M_y and M_z. However, in interprocedural optimization, as discussed earlier, we do not solve these constraints directly; instead, we propagate them up in the call graph. Assume now that a procedure p' calls p using `call p(X,X)`. As a result, the locality constraints above should be re-written in terms of X (i.e., they need to be re-mapped). That is, after the propagation, we re-write the constraints as:

$$M_x \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} u & 0 \\ u & 1 \end{pmatrix} \quad \text{and} \quad M_x \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} u & 0 \\ u & 1 \end{pmatrix}.$$

Now, solving M_x from these constraints is more difficult as M_x is involved in both the equalities. Intuitively, this is because the aliasing between Y and Z (introduced by the call to p) imposes more conditions on the data transformation to be found. This small example demonstrates how the locality constraints are propagated up in the call graph, and how the parameter arrays are translated after the propagation.

5. DETERMINING MEMORY LAYOUTS

It should be stressed that the locality constraints collected at the root should be solved for our unknowns; that is, the data transformation matrices. Obviously, it is possible that we may not have a solution for this large set of equalities. If this is the case, in order to find a solution, we need to sacrifice some constraints. In other words, we need to drop some constraints from further consideration, and try to solve the resulting (reduced) system of constraints again. This constraint elimination continues until we find a solution (i.e., determine all data transformation matrices).[2] To determine which constraints to drop, we use the weight components of the constraints. More specifically, we order the locality constraints according to their weights, and at each step we eliminate the constraint with the minimum weight (until the resulting system has a solution).

It should be noted that solving this set of constraints has additional complexities. Maybe the most challenging of these is the fact that the data transformation matrices should be non-singular. Therefore, in building a data transformation matrix, we should be careful. Note that each locality constraint can be expressed in terms of rows of the data transformation matrix. For example, Equation (1) can be re-written (for a t-dimensional array in an n-deep nest) as $\vec{m_{1x}}^T \vec{r_{nx}} = 0$, $\vec{m_{2x}}^T \vec{r_{nx}} = 0$, ..., $\vec{m_{(t-1)x}}^T \vec{r_{nx}} = 0$, and $\vec{m_{tx}}^T \vec{r_{nx}} = c$, where $\vec{m_{ix}}^T$ is the ith row of M_x and $\vec{r_{nx}}$ is the last column of R_x. To ensure non-singularity, we should be careful in selecting the rows $\vec{m_{1x}}^T$, $\vec{m_{2x}}^T$, ..., $\vec{m_{tx}}^T$. More specifically, these rows should be selected in such a way that they are linearly independent. To achieve this, in determining each row, we check whether this row is independent of the rows that have already been determined. Since data transformations do not affect data dependences, we do not need to check data dependences. Once the data transformation matrices have been determined, the output code can be generated using the approach proposed by Leung and Zahorjan [7].

[2] Note, however, that we only determine the transformation matrices for the arrays whose constraints are propagated to the root. The remaining data transformations (those that come from local arrays) are determined during the top-down phase when the associated procedures are visited.

6. TOP-DOWN PHASE: APPLYING LOOP TRANSFORMATIONS

After determining the memory layouts for the arrays, the next step (called the top-down phase) involves propagating down these layouts to the children nodes (see Figure 2(b) for an example case that consists of five procedures including `main()`). Let us assume for now that each procedure has a single parent (we will relax this requirement shortly). In receiving (inheriting) memory layouts from the parent of a procedure, we perform the following two steps (for the procedure in question):

- Apply loop transformations to satisfy these (inherited) layouts as much as possible.
- Determine memory layouts of the local arrays in the procedure.

Satisfying memory layouts through loop transformations means selecting a suitable loop transformation for each nest such that the nest exhibits good locality with the inherited layouts. Recall that a loop transformation T (whose inverse is T^{-1}) transforms the reference $(R_x, \vec{r_x})$ to $(R_x T^{-1}, \vec{r_x})$. If, however, the reference in question is first transformed using the data transformation matrix M_x, then, after the loop transformation, the reference becomes $(M_x R_x T^{-1}, M_x \vec{r_x})$. Since, at this point, M_x, R_x, and $\vec{r_x}$ are known (recall that M_x represents the data transformation that gives the inherited layout), we can determine T^{-1} for the best locality.

Specifically, we should select a T^{-1} such that

$$M_x R_x T^{-1} = \begin{pmatrix} u & u & u & \ldots & u & 0 \\ u & u & u & \ldots & u & 0 \\ u & u & u & \ldots & u & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ u & u & u & \ldots & u & c \end{pmatrix} \qquad (2)$$

is satisfied. As before, here, u denotes a *don't care* entry and c is a small value. Note, however, that unlike Equation (1), here we have two different choices for c. If we are able to select a T^{-1} such that c is 0, then this means we obtain temporal locality. On the other hand, if we can only find a T^{-1} such that c is a small non-zero constant (preferably 1), we have spatial locality. Obviously, as in the case of data transformations, it may or may not be possible to find such a T (or T^{-1}) matrix. Moreover, a suitable loop transformation matrix should satisfy multiple (array) references in the nest (possibly to different arrays).

Let us focus now on a given nest in a procedure p. Assume that this nest accesses s arrays, s' of which have already-determined memory layouts (i.e., non-local arrays). So, in the first step, we select a T^{-1} such that all references to these arrays are optimized (as explained above) either for temporal locality ($c = 0$) or spatial locality ($c = 1$). Once such a loop transformation is found for each nest, we build a second set of locality constraints to determine the memory layouts of the $s - s'$ arrays local to procedure p. More specifically, we need to find a data transformation matrix $M_{x'}$ for each such local array X' so that $M_{x'} R_{x'} T^{-1}$ exhibits good locality. Note that, here, $R_{x'}$ and T^{-1} are known, and $M_{x'}$ needs to be determined.

When a given procedure has two or more parents, the top-down phase becomes more difficult to handle. This is because different parents can require different memory layouts (at the end of the bottom-up phase) for the same array. If this array is also accessed

in the current procedure being optimized, then we need to decide which layout to consider in selecting loop transformations for the said procedure. Figure 2(c) presents an example call graph fragment depicting a procedure (p) with two parents (p' and p''). If, after the bottom-up phase, p' and p'' demand, say, column-major and row-major layouts, respectively, for the same array, we need to resolve this conflict in optimizing p.

There are at least two ways of resolving such a conflict. The first way exploits the weight components of the locality constraints as follows. It first calculates an *array weight*, which is simply the sum of the weights of the all constraints that involve the array in question. Then, in optimizing procedure p (during the top-down phase), we consider the layout of the array (instance) with the largest weight. In other words, depending on the weights (of the same array) in p' and p'', we select the memory layout to use in determining the loop transformations in p.

The second way of resolving such a conflict is to employ a compiler technique called *procedure cloning.* Cloning generates a new version of a procedure for specific interprocedural information (see [3] for more information on cloning). In the context of this paper, we can create different versions of a given procedure whenever multiple memory layouts (for the same array) *reach* the same procedure. For the call graph example in Figure 2(c), a typical cloning (assuming p' and p'' demand different layouts for the same array) creates two versions of p (denoted $p*$ and $p**$ in Figure 2(d)), one for each layout. That is, the loop transformations in $p*$ are applied assuming a column-major memory layout for the array while they are applied in $p**$ under a row-major memory layout. It should be stressed that procedure cloning increases the code space requirements (which is not desirable in embedded systems). So, in this paper, we adopted the first alternative.

7. DYNAMIC LAYOUT OPTIMIZATIONS

So far, we have assumed that each array will be assigned a single layout for its entire lifetime. In some cases, we might be able to achieve better performance by relaxing this constraint. As an example, consider the call graph in Figure 3. Let us focus on a single array and assume that each procedure demands the layouts (for that array) shown in the figure (RM means row-major and CM means column-major). One alternative for optimizing for this array is to select a row-major layout and then use loop transformations in three procedures that demand column-major layout. Such an optimization strategy may or may not be feasible depending on the data dependences in these procedures. Recall that loop transformations are bound by data dependences; that is, they must respect all data dependences in the nest.

An alternative optimization strategy is to use dynamic layout transformations; i.e., modifying the memory layout during the course of execution. Since both `Proc2()` and `Proc3()` demand column-major layout, a good point for converting layout (of the array in question) from row-major to column-major is at the beginning of `Proc2()`. Then, `Proc2()` and `Proc3()` can execute their codes with the column-major layout, and just before returning to `Proc1()`, we can transform the layout back to row-major. Obviously, there are many issues involved in implementing dynamic layout transformations. Maybe the most important issue is to decide whether to use dynamic transformations at all. Note that dynamic layout transformations are implemented by inserting extra code in the application to convert layouts (to copy the transpose of the original array into another). This entails extra code space, extra data space, and execution cycles. Therefore, a careful cost/benefit analysis is required. An equally important issue is to determine (in the call graph) the point at which to transform the layout. Yet a third issue is to decide whether to transform multiple arrays together. This might be useful if multiple arrays demand the same type of transformation. Answering these questions is beyond the scope of this paper; however, for comparison purposes, in the next section, we also report experimental results obtained using dynamic transformations. Our current approach does not handle recursive procedures or procedures that are called indirectly (e.g., by passing procedure-pointers); however, we plan to address these issues in a future study.

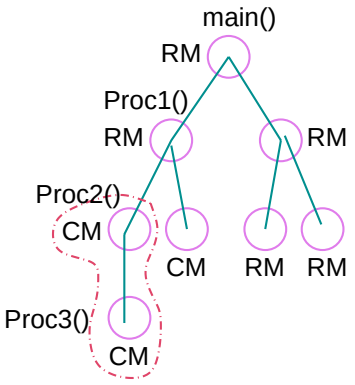

Figure 3: An example call graph with nodes marked with the layouts they demand.

8. EXPERIMENTS

To test the effectiveness of our approach, we used six array-intensive embedded applications. `Vcap` is a video capture and processing application. It generates video streams of different picture sizes, color spaces, and frame rates. `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. `Segt` is an image processing application that performs segmentation. `Face` is a face recognition algorithm. The input sets of these applications range from 211KB to 678KB. Their original execution times range from 57.7 seconds to 231.3 seconds. Our base cache architecture is 4KB, directed-mapped with a line size of 32 bytes. We use Dinero [4] to collect cache hit/miss statistics and then convert them to cycles. We assume a cache hit latency of 1 cycle and a miss latency of 60 cycles. All results presented below are percentage (execution time) improvements over the original (unoptimized) codes.

Figure 4 shows the performance (execution time) improvements over the original codes. The cache behavior trends (i.e., hit/miss behavior) are very similar to execution time trends, and thus omitted due to lack of space. `Loop` corresponds to a version that uses aggressive loop transformations for optimizing locality. Specifically, it uses the technique explained in Li's thesis [8]. This technique employs both linear transformations (e.g., loop permutation and scaling) and nonlinear transformations such as iteration space tiling. `Data` is an optimization strategy that employs only memory layout optimizations. It uses the approach proposed by Leung and Zahorjan [7]. `Loop+Data` is a technique that combines both loop and data transformations in a unified framework. The specific approach used in this paper is similar to that proposed in [5]. Finally, `Interprocedural` is the optimization technique discussed in this paper. Note that none of these versions employs dynamic layout transformation. All of these versions have been implemented using the SUIF compiler infrastructure from Stanford University [1]. From the results in Figure 4, we observe that interprocedural optimization is very successful and five of our six applications benefit from it. The average improvements due to

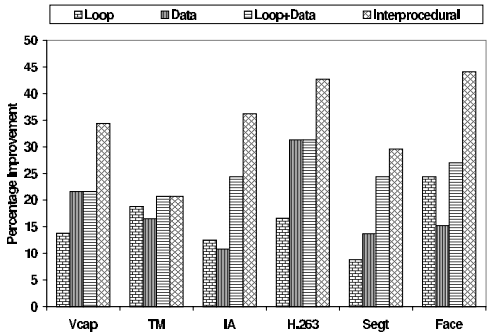

Figure 4: Percentage improvements.

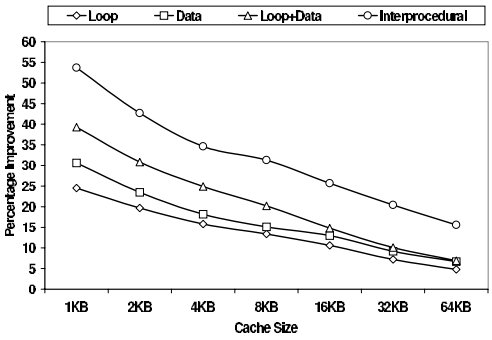

Figure 5: Impact of cache capacity.

`Loop`, `Data`, `Loop+Data`, and `Interprocedural` versions are 15.8%, 18.2%, 24.9%, and 34.6%, respectively. We also see from these results that neither `Loop` nor `Data` dominates the other. It should also be mentioned that `Loop+Data` used in our experiments represents the state-of-the-art in (intraprocedural) cache locality optimization for array intensive codes.

To see the impact of the cache capacity on performance improvements, we performed another set of experiments where we changed the cache size from 1KB to 64KB. The results given in Figure 5 represent the averages across all benchmarks and indicate that the `Inter procedural` version outperforms the other versions in all cache sizes. While, as expected, the savings due to our approach diminish as we increase the cache size (this is because the original code captures more locality with a larger cache size and performs better), we observe that even with a 64KB cache we obtain more than 15% improvement.

In our next set of experiments, we studied the impact of dynamic layout transformations on performance. We conducted experiments

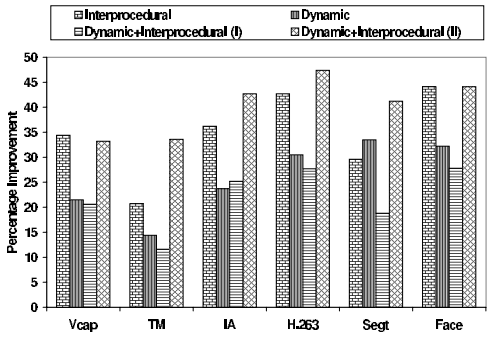

Figure 6: Percentage improvements.

with four different versions and present the improvements in execution times in Figure 6. `Inter procedural` is the version discussed in this paper. `Dynamic` is a dynamic layout optimization strategy that does not perform any interprocedural analysis. `Dynamic- Inter procedural (I)` is a version that combines interprocedural optimization with dynamic layouts. It uses dynamic layouts aggressively; that is, whenever two procedures (neighboring in the call graph) demand different layouts for the same array, a dynamic layout transformation is performed when transitioning from the caller to the callee. Finally, `Dynamic- Inter procedural (II)` is similar to `Dynamic- Inter procedural (I)` except that it uses dynamic layout transformations more carefully. Specifically, it does not use dynamic layouts unless two successively activated procedures use the transformed layout before it is transformed to back to its original form. Both `Dynamic- Interprocedural (I)` and `Dynamic- Inter procedural (II)` are hand optimizations. The percentage improvements (i.e., across all six benchmarks) for `Inter procedural`, `Dynamic`, `Dynamic- Inter procedural (I)`, and `Dynamic- Interprocedural (II)` versions are 34.6%, 26.0%, 21.9%, and 40.4%, respectively. These results reveal that it is not a good idea to use dynamic layouts indiscriminately. They also show that employing dynamic layout optimizations without interprocedural analysis does not generate the best results.

9. CONCLUDING REMARKS

One of the main problems in extracting maximum performance out of current embedded architectures is poor data cache locality. Due to cost and space concerns, increasing cache capacity arbitrarily is not a long-term solution. So, software solutions are a promising alternative. This paper extends the state-of-the-art in cache locality optimization by showing how locality can be optimized interprocedurally. Our results show that the interprocedural optimization is much more effective than the previous locality-based techniques that handle each procedure in isolation, and that proper use of dynamic optimization makes a difference.

10. REFERENCES

[1] S. P. Amarasinghe et al. The SUIF compiler for scalable parallel machines. In *Proc. the Seventh SIAM Conf. on Parallel Proc. 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] K. D. Cooper et al. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993, pages 105–118.

[4] Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/∼markhill/DineroIV/

[5] 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.

[6] M. Kandemir et al. A framework for interprocedural locality optimization. In *Proc. Intl. Conference on Parallel Processing*, Aizu, Japan, 1999.

[7] 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.

[8] W. Li. *Compiling for NUMA Parallel Machines*. Ph.D. Thesis, Cornell University, Ithaca, New York, 1993.

[9] 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.

[10] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM Conf. Programming Language Design and Implementation*, June 1991.

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