Strategies for Improving Data Locality in Embedded Applications[*]

N. E. Crosbie[†] M. Kandemir[‡] I. Kolcu[§] J. Ramanujam[¶] A. Choudhary[||]

Abstract

This paper introduces a dynamic layout optimization strategy to minimize the number of cycles spent in memory accesses in a cache-based memory environment. In this approach, a given multi-dimensional array may have different memory layouts in different segments of the same application if doing so improves data locality (cache behavior) beyond the static approaches that fix memory layouts at specific forms at compile-time. An important characteristic of this strategy is that different memory layouts that a given array will assume at run-time are determined statically at compile-time; however, the layout modifications (transformations), themselves, occur dynamically during the course of execution. To test the effectiveness of our strategy, we used it in optimizing several array-dominated applications. Our preliminary results on an embedded MIPS processor core show that this dynamic strategy is very successful and outperforms previous approaches based on loop transformations, data transformations, or integrated loop/data transformations.

1 Introduction

Many embedded image and video processing applications operate on large multi-dimensional arrays of signals using multi-level nested loops. An important characteristic of these codes is the regularity in data accesses, which might be exploited using an optimizing compiler for improving cache memory performance. For example, the compiler can permute the loops in a given multi-level nest to exploit spatial and/or temporal reuse in the innermost loop positions. Alternatively, the memory layouts of arrays can be modified to make the data organization in memory compatible with the loop access pattern. Previous research (e.g., [3, 10, 2]) shows that loop (iteration space) oriented and data space (memory layout) oriented transformations are complementary.

Previous approaches to layout optimizations are referred to as static approaches, meaning that there is a single memory layout associated with each array variable and that this layout is valid (fixed) throughout execution. Because a data layout that is good for one segment of code may not be good for another segment, dynamic layout transformations (i.e., associating more than one layout with an array variable and switching between different layouts as the code executes) offers a potentially better alternative.

In this paper, we present and evaluate a compiler-directed dynamic layout optimization strategy. Our strategy works on a nest flow graph representation of the code and specifies memory layouts for each array in each nested loop. In this strategy, the compiler determines the layouts statically at compile time and inserts dynamic layout conversion code in the application. However, the layout conversions themselves are activated at runtime and layouts are transformed automatically as the application executes. Consequently, an array can assume different layouts in the course of execution. The objective here is to reduce the number of cycles spent in memory stalls; that is, improving data locality and exploiting on-chip memory space as much as possible. To test the effectiveness of our strategy, we used it in optimizing several array-dominated applications. Our preliminary results show that this dynamic strategy is very successful and outperforms previous approaches based on loop transformations, data transformations, or integrated loop/data transformations.

The rest of this paper is organized as follows. Section 2 introduces components of our framework. Section 3 presents our optimization strategy which uses both dynamic layouts and loop transformations. Section 4 presents our experimental methodology and reports preliminary results. Finally, Section 5 concludes the paper with a summary.

2 Components of Our Framework

Our framework employs a static, integrated locality optimization technique as the key component and uses this static technique recursively to determine the best combinations of

[*] This work was supported in part by the NSF CAREER Award 0093082.

[†] CSE Department, Pennsylvania State University, University Park, PA 16802, USA. Email: crosbie@cse.psu.edu.

[‡] **Corresponding Author.** CSE Department, Pennsylvania State University, University Park, PA 16802, USA. Email: kandemir@cse.psu.edu.

[§] Department of Computation, UMIST, Manchester M60 1QD, UK. Email: I.Kolcu@student.umist.ac.uk.

[¶] ECE Department, Louisiana State University, Baton Rouge, LA 70803, USA. Email: jxr@ee.lsu.edu.

[||] ECE Department, Northwestern University, Evanston, IL 60208, USA. Email: choudhar@ece.nwu.edu.

0-7695-1441-3/02 $17.00 © 2002 IEEE

local loop transformation and local memory layouts. What we exactly mean by local will become clear later in the paper. In this static approach, the key to optimizing cache locality is to apply loop transformations for exploiting one type of locality and data transformations for a different type of locality. Specifically, the existing loop transformation theory [14] is specialized for optimizing temporal locality. To achieve this, the compiler needs to detect the amount of potential temporal locality in a loop nest and to come up with a suitable loop transformation matrix to exploit that amount. After this process, the references within the loop nest are divided into two groups: the ones with optimized temporal locality, and the ones that do not exhibit temporal locality. For the latter group of references, we apply data layout transformations to enhance their spatial locality.

To handle the case where there are multiple independent nested loops, the static approach uses profile data and optimizes one nest at a time, starting with the most costly (most important) nest. After optimizing each nest, new layouts are determined (for arrays whose layouts have not been determined so far) and these new layouts along with the old ones are propagated to the remaining (yet to be optimized) nests. In other words, the static optimization technique can be made to target at one, two, or more nests at a time. This framework can employ row-major and column-major layouts, higher-dimensional equivalents of them, and more general linear layouts such as diagonal layouts. Further details of the static approach are beyond the scope of this paper and can be found in [7].

Our dynamic optimization technique also employs a cache miss estimation technique. Our current implementation uses the approach discussed in Carr et al. [4] with three modifications. First, unlike the original approach, the enhanced version can estimate the number of misses when different arrays have different memory layouts. Second, the write misses are also included in miss calculation. Third, we also take the contribution of conflict misses into account using the technique proposed by Sarkar et al. [13]. The details of cache miss estimation techniques can be found in references [4, 13]. We are working on integrating the cache-miss-equation technique due to Ghosh et al. [6] into our framework.

3 Dynamic Layout Optimization

3.1 Nest Flow Graph

Our optimization strategy uses a procedure representation called Nest Flow Graph (NFG). An NFG is a directed graph $G(V, E)$ where each node $v_i \in V$ represents a nest, and a directed edge $e_{ij} = (v_i, v_j)$ from v_i to v_j indicates that there exists a flow of control from the nest represented by v_i to the nest represented by v_j.[1] Note that this control flow might be because of an explicit jump (to v_j) from within or immediately after the nest represented by v_i, or because of the fact that v_j immediately follows v_i. In a sense, an NFG is a coarse-grain control flow graph (CFG), the difference being that each node in an NFG corresponds to a nest instead of a basic block (as in the case of CFG).

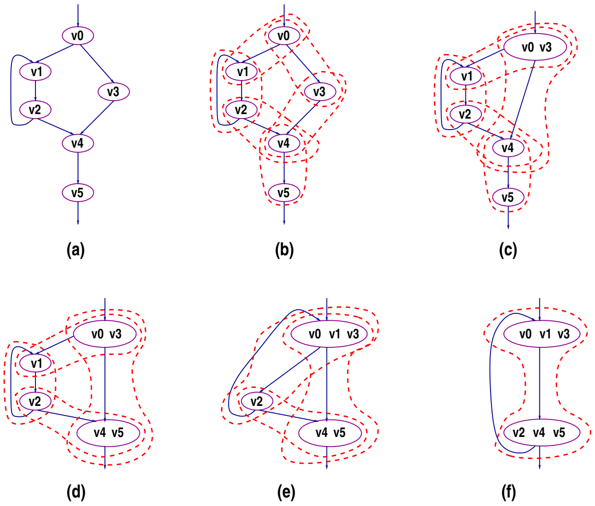

Figure 1. (a) An example NFG (nest flow graph). (b-f) Clusters at different levels.

Figure 1(a) shows an NFG which consists of six separate loop nests, denoted using v_0, v_1, v_2, v_3, v_4, and v_5. Note that a path in the NFG indicates a possible execution order for nests. The two nodes involved in a cycle in NFG (e.g., v_1 and v_2 in Figure 1(a)) indicate that there is a loop other than for-loop (e.g., constructed using a while loop or explicit jumps) in the code that encloses the corresponding nests. While it is also possible to build a weighted NFG where the NFG edges are annotated estimated execution frequencies (obtained through static compiler analysis or profile data where available), in this paper, we employ an unweighted NFG. It should be noted, however, that adopting a weighted NFG can greatly impact the success of our dynamic optimizer. We intend to employ a weighted NFG in our future work on this topic.

It is important to note that the nodes in the NFG represent perfectly-nested nests (i.e., the nests where all non-loop statements reside in the innermost position). A pre-processing step in our compiler uses loop distribution, loop fusion, and code sinking [14] to convert imperfectly nested loops to perfectly-nested one.

[1] In the remainder of this paper, we use v_i to refer to both a node in the NFG and the corresponding nest in the code.

3.2 Iterative Algorithm

Our dynamic optimization algorithm starts by calling the static optimizer for each nest separately. The static optimizer then determines for each nest a suitable loop transformation and accompanying memory layouts for the arrays referenced in it. It also inserts explicit layout conversion loops between nests. As an example, consider again the NFG given in Figure 1(a). Let us assume that after optimizing v_0 we find that, for two arrays accessed in this nest, the best memory layouts are row-major and column-major. Assume further that the corresponding layouts for the same arrays in v_3 are column-major and diagonal, respectively. Consequently, we need to transform (using layout conversion loops) the layout of the first array from row-major to column-major and that of the second from column-major to diagonal between these two nests (i.e., along the edge (v_0, v_3) in Figure 1(a)). Depending on the dimentionalities and sizes of the arrays, these conversions can be done using a single nest or two separate nests (one per array).

It should be noted that at this point we have an optimized code of fine-granularity; that is, each nest works with the best memory layouts (and loop transformation) when considered in isolation. Such layouts and loop transformations are called *local* in this paper as they are obtained considering only a single nest, without taking into account the coupling between nests (due to common arrays accessed). While such a fine-granularity scheme can generate the best result when nests are considered in isolation, the cost of layout conversion loops can easily offset the potential benefits. Note that layout conversion loops incur two major overheads. First, they consume precious CPU cycles in copying elements from one array to another (as explained later in the paper). In particular, in machines without fast memory-to-memory support, such copies can be quite costly. Second, they incur a memory overhead. This last overhead manifests itself in two ways. First, a significant number of cache misses can be experienced during copying itself. Second, copying can pollute the cache. This is particularly true if the array with the new layout is not used immediately after copying. Because of these reasons, it might be a good idea to perform as few explicit layout conversions as possible. What this means is that, whenever possible, the same layout should be used for the same array. Therefore, there is an tradeoff between local optimization (best local layouts, and layout transformations with conversion overhead) and global optimization (no conversion overhead but potentially locally suboptimal layouts). Note that the extreme case (coarse-granular optimization) with a single layout per array throughout the program execution corresponds to the static optimizer (when targeted the entire procedure).

The objective of our optimization strategy is to strike a balance between the local optimizations and the global view. That is, on one hand, we try to make sure that each nest executes with the best possible combination of memory layouts and loop order; on the other hand, we would like to minimize the number of layout conversion loops.

Next, we informally describe how we achieve this objective. We start with the fine-granular code and try to eliminate the conversion loops iteratively (only) if doing so improves the overall (procedure-wide) performance. To eliminate the conversion loops, our strategy is to increase the scope of local optimization. To illustrate the approach using a specific example, let us consider the NFG depicted in Figure 1(a) once more. Let us also assume that we have already run the static optimizer for each nest and the layout conversion loops have been determined. In the next step, our dynamic strategy clusters the nests into groups of two. Two nests (two nodes) v_i and v_j are placed into the same cluster (named v_iv_j) if there exists an edge (v_i, v_j) between them. Note that a given nest may belong to multiple clusters. After that, the number of cache misses for each cluster v_iv_j is estimated. Then, we focus on the cluster with the maximum number of misses. It should be emphasized that the misses in a cluster v_iv_j has three components: (i) misses due to v_i (denoted m_i); (ii) misses due to v_j (denoted m_j); and (iii) misses due to the layout conversion loop(s) between v_i and v_j (denoted $m_{i\rightarrow j}$). Note that, as far as the compiler's view of the code is concerned, this cluster is the one which is responsible from the largest portion of the overall cache misses of the application (after the application of the static optimizer). Next, we optimize this cluster, which consists of two nests, using the static optimizer but considering the original (untransformed) forms of the nests and layouts. Using the miss estimator, we then calculate the number of misses of this optimized cluster, m_{ij}. Subsequently, we check if the following condition holds:

$$m_{ij} < m_i + m_j + m_{i\rightarrow j} \qquad (1)$$

If it does, what this means is that it is better to consider v_i and v_j together as a cluster and optimize them using the static optimizer, instead of optimizing each of them separately and using layout conversion loop(s) between them. The compiler then updates the NFG in question to reflect this fact and applies the same strategy recursively to the updated NFG. Returning to our example, Figure 1(b) shows the clusters for the NFG in Figure 1(a). Let us assume now that the cluster v_0v_3 is the one with the highest number of misses. Consequently, using the expression (1), the compiler checks whether $m_{03} < m_0 + m_3 + m_{0\rightarrow 3}$ holds true. If so, v_0 and v_3 are optimized together as a cluster (using the original forms of the nests). It then updates the NFG as follows. v_0 and v_3 are combined into a single node, and all incoming (outgoing) edges to (from) the original nodes v_0 and v_3 are connected to the new node (denoted v_0v_3). The dynamic optimizer proceeds recursively by considering this updated NFG. It first clusters the nodes (including the new combined node) as shown in Figure 1(c) and computes the number of misses for each cluster. Let us assume that v_4v_5 is the new cluster with the maximum number of cache misses. The compiler then checks whether the expression $m_{45} < m_4 + m_5 + m_{4\rightarrow 5}$ holds true. If it does, nodes v_4 and v_5 (the original nests) are optimized together, the NFG is updated, and the optimization process proceeds recursively. Figures 1(d) through (f) show a possible progress of our dynamic optimizer.

There are several important points to note here. First, if at any point in the program, the inequality (1) is not satisfied, the compiler considers the cluster with the second highest number of misses and checks the validity of the inequality (1) for that cluster, and so on. If, when working on a given NFG, the inequality given by (1) is not satisfied for any cluster, then the dynamic optimization algorithm terminates and returns the current NFG with transformed arrays and loops as output. Second, when computing the number of misses for a cluster, we always consider the original forms of the nests involved and optimize them collectively using the static optimizer summarized earlier. Third, it should be noted that, depending on the degree of coupling between nests, different loop transformations might be selected when a given nest is optimized individually and when it is optimized as part of a cluster. This is because the nests in a given cluster are, in general, coupled as they might manipulate common arrays. Recall that the static optimizer finds both the loop transformations and (static) memory layouts which are most suitable for a given code fragment (which might consist of a single or multiple nests), and when multiple nests are involved, it takes coupling between nests into account by propagating memory layouts between them. A sketch of the dynamic optimizer is given in Figure 2. The algorithm stops when the variable progress remains false during an entire execution of the innermost loop.

```
INPUT: An array-dominated procedure
OUTPUT: Optimized procedure with transformed loops
     and memory layouts

begin
 pre-process code to convert imperfectly-nested loops
     to perfect nests;
 build NFG;
 for each nest $v_i$ do
  optimize $v_i$ using the static optimizer;
  estimate the number of misses of $v_i$ (denoted $m_i$);
 endfor
 progress = true;
 while(progress is true) do
  progress = false;
  L: cluster the current NFG into groups of two;
  order the clusters in a set S according to decreasing value
     of $m_i + m_j + m_{i\rightarrow j}$;
  for each cluster $v_iv_j$ in S (starting from top) do
   optimize the original forms of $v_i$ and $v_j$ together
     using the static optimizer;
   compute the number of misses $m_{ij}$;
   if ($m_{ij} < m_i + m_j + m_{i\rightarrow j}$) then
      update the NFG;
      progress = true;
      goto L;
   endif;
  endfor;
 endwhile;
 insert the layout conversion loops and optimize them;
end
```

Figure 2. **Dynamic locality optimizer.**

3.3 Conversion Code Placement

An important issue in implementing dynamic layout optimizations is deciding where to place the code to change from one layout to another (i.e., layout conversion loops). As mentioned earlier, a conversion loop changes an array layout from one form to another. In our current approach, for each array to be layout-transformed, we use two arrays to implement this idea. The first of these is the original array whose memory layout is to be transformed and the second one is the new array which will represent the layout transformed version of the original array.

There are two objectives in selecting the point to insert the layout conversion code (nest). First, we would like to minimize the number of conversion codes. This is because the conversion code is pure overhead and should be optimized away as much as possible. In our current implementation, we achieve this by (i) not inserting a conversion code unless it is necessary and (ii) trying to perform multiple layout conversions (associated with multiple arrays) using the same conversion nest. The second objective is to place the conversion code as late as possible in the code. The reason for this is that it might be possible that (because of the flow of control imposed by conditional program constructs) a conversion code would not be executed at all. It should be noted that, in some cases, placing the conversion code early in the application code might result in this code being unnecessarily executed.

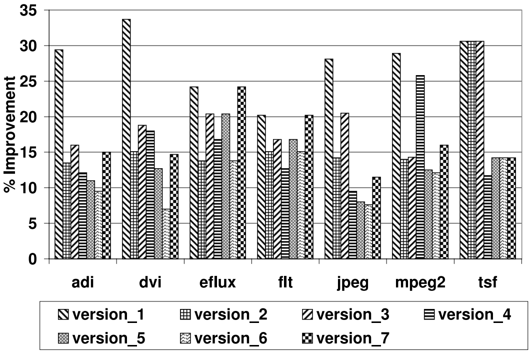

Figure 3. **Percentage execution time improvements for different versions.**

4 Experiments

4.1 Implementation and Methodology

To evaluate the effectiveness of our approach and compare it to other compiler-based techniques for improving locality, we used seven complete application codes. `adi` is an alternate direction integration code; `dvi` is a speech decompression algorithm; `elux` and `tsf` are two benchmark codes from Perfect Club; `flt` is a multi-phase filtering algorithm; `jpeg` is an image compression code; and finally, `mpeg2` is a player for MPEG-2 video bit-streams.

We used seven versions of each application code. $version_1$ is the code obtained using the dynamic optimization strategy discussed in this paper. $version_2$ is the fine-granular approach to dynamic optimization, where each nest is optimized individually and the layout conversion loops are inserted between nests whenever necessary. $version_3$ is the other extreme where the entire procedure is optimized as a single monolithic code using both loop and data layout transformations. This is the output of the static optimizer discussed in [7] (when targeted the entire code). Note that all these three versions use both loop and data layout transformations. $version_4$ is the result of classical loop optimizer which uses both unimodular loop transformations (e.g., loop permutation) and non-unimodular loop transformations such as tiling and unrolling. The details of this version can be found in [9]. Note that this version does not employ any data transformation. The remaining versions use only data (memory layout) transformations and no loop transformations. $version_5$ is similar to $version_3$ except that it does not use any loop transformation. It is a static approach and selects a single layout for each array considering the whole procedure at once. $version_6$ is the fine granular form of $version_5$. It selects the optimal layouts for each nest and inserts layout conversion code between nests (where necessary). It is different from $version_2$ in the sense that it does not apply any loop transformation. Finally, $version_7$ is the layout transform-only version of our approach ($version_1$). Note that in all versions, the layout conversion loops (if any) are optimized exactly the same way; therefore, the differences between performances of different versions are due to respective high-level optimization strategies.

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

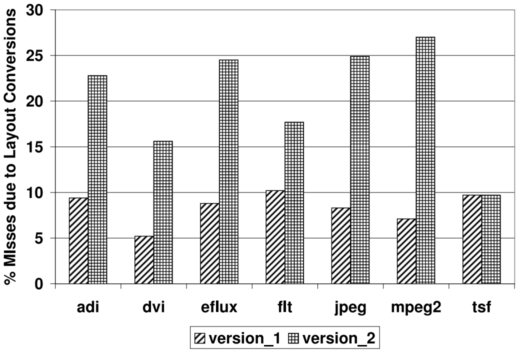

Figure 4. **Contribution of conversion code misses.**

4.2 Results

Figure 3 shows the execution time improvements brought about by different versions over the original code without any optimization. We observe that average improvements due to $version_1$ through $version_7$ are 27.9%, 16.6%, 19.6%, 15.2%, 13.7%, 11.3%, and 16.5%, respectively, showing that our dynamic optimization strategy outperforms the other versions. In fact, we see that, except for one case (`tsf`), our strategy generates the best results for all benchmarks. In `tsf`, our approach, the coarse-granular (procedure-wide single layout), and the fine-granular strategy generated the same result. It can also be seen that in two codes, namely, `eflux` and `flt`, the loop transformation part of our strategy did not bring any additional benefit over the data layout transformation part. Therefore, in these two codes, $version_1$, $version_2$, and $version_3$ resulted in the same code as $version_7$, $version_6$, and $version_5$, respectively. In the remaining benchmarks, we observe a clear superiority of our optimization strategy over others. We also observe that $version_3$ performed better than $version_2$. This is due to the excessive performance bottleneck exhibited by the layout conversion codes in the fine-granular optimization strategy. It should also be mentioned that the cache miss improvements due to $version_1$ to $version_7$ are 34.4%, 23.0%, 26.1%, 20.4%, 21.4%, 17.8%, and 21.1%, respectively.

To study this last point further, Figure 4 gives the percentage of overall cache misses due to layout conversion loops in $version_1$ and $version_2$. It can be seen that in $version_1$ these misses constitute only 8.4% (on an average) whereas the corresponding number for $version_2$ is 20.3%. This result indicates that it is extremely important to adopt a global view in layout-optimizing a given code.

As discussed in previous section, it might also be critical to optimize the layout conversion loops as much as possible. To evaluate this issue quantitatively, we also experi-

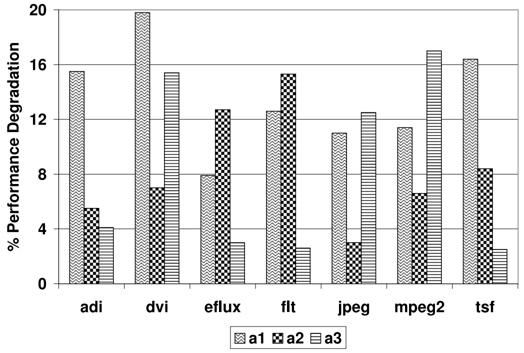

Figure 5. Performance degradation due to less optimized conversion code placement.

mented with three alternate (less optimized) conversion loop placement strategies. Recall that our default conversion code placement method is highly optimized, meaning that the conversion loops have been placed as late as possible (in the application code) unless this leads to an extra increase in code size, multiple conversions (for different arrays) have been performed using a single nest if it is possible to do so, and the conversion loops themselves have been optimized using loop tiling as much as possible. Figure 5 gives the percentage (execution time) degradations (for $version_1$) when we use less-optimized alternatives. Alternative `a1` gives percentage degradation when the conversion loops are not optimized using iteration space tiling. The results indicate that using tiling is critical, and not using it degrades performance by 13.5%, on average. `a2` shows percentage degradation when conversion loops are placed as early as possible instead of our default placement, which is as late as possible. We observe a 8.4% performance degradation on average. Finally, alternative `a3` gives the degradation in performance when no conversion loop reuse is employed, i.e., when separate conversion loops are used for each layout-transformed array. We can see that the average performance degradation in this case is 8.2%. Overall, based on these results, we can conclude that all three aspects of our conversion code placement strategy are critical.

5 Conclusions

Researchers working on hardware and software systems have made a number of important recent advances in memory optimizations. One of these is the use of techniques for enhancing data locality enhancing techniques to improve program performance. Previous compiler-based techniques to optimizing data locality are either pure loop-oriented strategies, pure layout-based strategies, or integrated (loop plus data layout) strategies that use static (fixed) program-wide data layouts for arrays. In this paper, we presented a dynamic optimizer using which it is possible to obtain further performance gains over current techniques.

References

[1] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, edited by A. Nicolau et al., MIT Press, 1991.

[2] 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.

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

[4] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, October 1994.

[5] S. Coleman, and K. McKinley. Tile size selection using cache organization and data layout. In *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[6] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. ACM International Conference on Supercomputing,* Vienna, Austria, July 1997.

[7] 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 Micro-architecture,* Dallas, TX, December 1998, pp. 285–296.

[8] 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.

[9] W. Li. Compiling for NUMA parallel machines. *Ph.D. Thesis,* Computer Science Department, Cornell University, Ithaca, NY, 1993.

[10] M. O'Boyle and P. Knijnenburg. Integrating loop and data transformations for global optimisation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Paris, France, October 1998.

[11] P. Panda, N. Dutt, and A. Nicolau. SRAM vs. data cache: The memory data partitioning problem in embedded systems. *Technical Report 96-42,* University of California, Irvine, CA, September 1996.

[12] P. Panda, N. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 18, No. 1 January 1999.

[13] V. Sarkar, G. R. Gao, and S. Han. Locality analysis for distributed shared-memory multiprocessors. In *Proc. the Ninth International Workshop on Languages and Compilers for Parallel Computing,* Santa Clara, California, August 1996.

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

[15] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling, In *Proc. the 36th ACM/IEEE Design Automation Conference,* June 21–25, 1999, New Orleans, LA, pp. 811–816.