

Code Restructuring for Improving Cache Performance of MPSoCs

G. Chen and M. Kandemir

Computer Science and Engineering Department
Pennsylvania State University, University Park, PA 16802, USA
{guilchen,kandemir}@cse.psu.edu

Abstract—One of the critical goals in code optimization for MPSoC architectures is to minimize the number of off-chip memory accesses. This is because such accesses can be extremely costly from both performance and power angles. While conventional data locality optimization techniques can be used for improving data access pattern of each processor independently, such techniques usually do not consider locality for shared data. This paper proposes a strategy that reduces the number of off-chip references due to shared data. It achieves this goal by restructuring a parallelized application code in such a fashion that a given data block is accessed by parallel processors within the same time frame, so that its reuse is maximized while it is in the on-chip memory space. This tends to minimize the number of off-chip references since the accesses to a given data block are clustered within a short period of time during execution. Our approach employs a polyhedral tool that helps us isolate computations that manipulate a given data block.

I. INTRODUCTION

Multi-Processor-System-on-a-Chip (MPSoC) architectures are becoming a promising choice in designing of complex embedded systems [15]. There are several reasons for this trend. First, these architectures employ simpler easy-to-verify processors as compared to complex single processor based architectures. This makes them arguably easier to validate and verify. Second, an MPSoC architecture can be clocked at a reduced speed (again compared to single processor based systems), and this can improve power efficiency. This flexibility is a direct result of having multiple processors on the same die; that is, the performance loss due to reduced clock frequency can be compensated through on-chip parallel processing. Third, these architectures match very well with high level (e.g., loop level) parallelism and this makes them perfect candidates to execute array/loop-intensive embedded applications from different domains. For example, many applications from the domain of embedded multimedia processing are built from series of nested loops accessing multi-dimensional arrays of signals. Several recent studies such as [8] discuss the benefits of MPSoCs in the context of array/loop-intensive applications.

One of the critical components of an MPSoC based system is its memory hierarchy. While circuit level optimizations and architectural issues are very important (e.g., how many levels the memory hierarchy has, whether it is software or hardware managed, what the unit of transfer between memory components is, how interprocessor synchronization is maintained, etc), equally important are software level optimizations that minimize the execution cycles spent in memory accesses and/or reduce memory energy consumption. Recent research in literature [13] considered both hardware and software techniques for improving memory behavior of MPSoCs.

An important way of improving memory performance of MPSoC architectures is to improve data locality and minimize non-local data accesses. In particular, in an MPSoC architecture, off-chip memory accesses can be extremely costly from both performance and energy viewpoints, and the number of such accesses should be minimized as much as possible. Figure 1 illustrates a typical memory hierarchy for an MPSoC. The first level is the private on-chip memory (L1) of each processor, and is the first pick from both performance and energy viewpoints. If however we miss in this memory, our next choice would be finding the data in the shared on-chip memory (L2). While such an access is costlier than accessing the data from the L1 memory, it is still better than going to the off-chip memory. This argument, while it is certainly true from the execution cycles' viewpoint, it is even more true from the power consumption perspective. This is because an off-chip access means exercising large

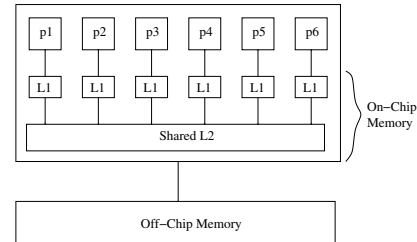


Fig. 1. High-level view of an MPSoC with its off-chip memory.



Fig. 2. Overall framework of our approach.

off-chip interconnect and accessing a memory with a large switch capacitance. While the performance overhead can, in some cases, be partially hidden by concurrent on-chip processing, power overheads cannot be. Based on this discussion, the goal of a compiler-directed memory optimizations should be minimizing the number of off-chip accesses even if this means increasing the number of on-chip traffic. The strategy proposed in this paper achieves this goal by restructuring a parallelized application code in such a fashion that a given data block is accessed by parallel processors within the same time frame, so that its reuse is exploited while it is in the shared on-chip memory components (L1 or L2 in Figure 1). This tends to minimize the number of off-chip references since the accesses to a given data block are clustered within a short period of time during execution. Note that ensuring data locality for each processor in isolation (i.e., optimizing intra-processor data locality only) does not guarantee locality for shared data (i.e., the inter-processor data locality). In particular, if the reuse distance (i.e., the gap between two successive uses) for shared data is large, each processor may have to go to the off-chip memory for the same data element. Our main goal is to reduce reuse distances for shared data elements as much as possible.

This paper presents an automated code restructuring (loop iteration reorganization) strategy that achieves this for parallelized array/loop-intensive applications running on cache memory based MPSoC architectures. This strategy employs a polyhedral tool that helps us isolate the set of loop iterations that manipulate a given data block. Using this polyhedral tool, our approach structures the local iteration space of each processor in such a way that the reuse distance for the shared data is reduced.

The rest of this paper is organized as follows. The details of our compiler-directed approach are presented in Section II. Section III discusses related work. Finally, our concluding remarks are presented in Section IV.

II. OUR APPROACH

A. Overview of Our Approach

The overall framework of our approach is given in Figure 2. We first use a scheduling algorithm to schedule data accesses such that reuse distances for the shared data are reduced. Then, we use a convex set building tool available in the Omega Library [2] to group the iteration sets into convex regions. For each convex region, the code

TABLE I
NOTATION USED IN OUR PRESBURGER FORMULATION.

Notation	Explanation
$\prec, \preceq, \succ, \succeq$	Lexicographical ordering on vectors
$ S $	Number of elements in set S
P	Number of processors
p	A processor id
\vec{l}	A loop iteration point (iteration vector)
\vec{a}	A data element index (array/subscript index)
R	An array reference
\mathcal{R}	Set of array references in a loop nest
$\vec{l}_{s,p}, \vec{l}_{e,p}$	Lower and upper bounds for set of iterations assigned to a processor p
\vec{d}_l, \vec{d}_e	Array lower and upper bounds
$\mathcal{D}(\vec{a})$	Data tile represented by data element \vec{a}
$\mathcal{J}_p(\mathcal{D}(\vec{a}))$	Set of iterations assigned to processor p and access $\mathcal{D}(\vec{a})$

generation tool of the Omega Library can generate one loop nest to iterate over all the iteration sets in this region. Finally, we use the Omega library to generate code that enumerates the iterations within each iteration set. The details of these three phases will be explained in the rest of this paper.

We use Presburger formulation [3] to formulate our problem within polyhedral algebra. Presburger formulation is a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives (\vee , \wedge , and \neg), and the existential and universal quantifiers (\exists and \forall). In this work, we employ the Omega Library, a polyhedral tool, to manipulate integer tuple relations and sets, which are described using Presburger formulas. Table I gives the notation used in our formulation of the code restructuring problem. We use vectors to represent loop iterations as well as array data elements accessed (i.e., their indices). As an example, for a loop nest where i is the outer loop and j is the inner loop, the vector $(i \ j)^T$ represents different iterations for the different values of i and j . A vector $(i+1 \ j)^T$, on the other hand, represents the different array elements accessed by an array reference $X[i+1][j]$ for the different values of i and j .

For now, we focus on the single array case (i.e., each loop nest accesses a single array); we will discuss the multiple array case shortly. We assume that an array space is logically divided into multiple *data tiles*. All data tiles are of equal size except possibly at boundaries of the array space. Each tile is identified by the position of its first element. Specifically, we use $\mathcal{D}(\vec{a})$ to denote the tile (i.e., the set of elements) represented by data element \vec{a} . In mathematical terms, we have:

$$\mathcal{D}(\vec{a}) = \{\vec{d} \mid \vec{a} \preceq \vec{d} \preceq \vec{a} + \vec{l}\},$$

where \vec{l} keeps the size of a tile in each array dimension.

Using Presburger arithmetic, we can also capture the set of loop iterations that are assigned to processor p (as a result of loop parallelization) and access the elements in a given data tile $\mathcal{D}(\vec{a})$. That is,

$$\mathcal{J}_p(\mathcal{D}(\vec{a})) = \{\vec{l} \mid \vec{l}_{s,p} \preceq \vec{l} \preceq \vec{l}_{e,p} \wedge \exists R \in \mathcal{R} : R(\vec{l}) \in \mathcal{D}(\vec{a})\}.$$

Assume that there are q tiles, and the j th tile is represented by data element indexed by \vec{a}_j . At a high level, our approach can be summarized as reordering the set of loop iterations assigned to each processor such that the iteration sets that access the same data tile are scheduled at the same time as much as possible. For example, all the following *iteration sets* should be scheduled concurrently:

$$\mathcal{J}_{p_1}(\mathcal{D}(\vec{a}_1)), \mathcal{J}_{p_2}(\mathcal{D}(\vec{a}_1)), \mathcal{J}_{p_3}(\mathcal{D}(\vec{a}_1)), \dots, \mathcal{J}_{p_P}(\mathcal{D}(\vec{a}_1)),$$

where P is the number of processors in our MPSoC, and we should repeat this for all other \vec{a}_i s. This execution sequence (schedule) is depicted in Figure 3. Note that the order of the different layers could be different as long as the iteration sets accessing the same data tile are scheduled at the same time.

An important issue that needs to be addressed at this point is the possibility of two different sets, e.g., $\mathcal{J}_p(\mathcal{D}(\vec{a}_i))$ and $\mathcal{J}_p(\mathcal{D}(\vec{a}_j))$, accessing the same data element. For example, it is possible that for a $\vec{l} \in \mathcal{J}_p(\mathcal{D}(\vec{a}_i))$, there is a reference R such that $R(\vec{l}) \in \mathcal{D}(\vec{a}_j)$, where $i \neq j$. This is possible because a given iteration can access multiple data elements when there are multiple references to the array within

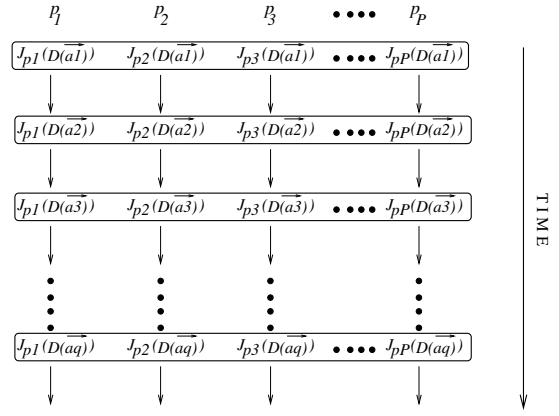


Fig. 3. Schedule order determined by our approach for a case with P processors and q tiles.

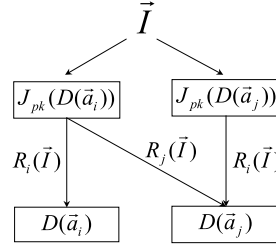


Fig. 4. A case where an iteration belongs to more than one iteration set.

the loop. As a consequence, $\mathcal{J}_p(\mathcal{D}(\vec{a}_i)) \cap \mathcal{J}_p(\mathcal{D}(\vec{a}_j))$ is not necessarily empty for all i s and j s. Therefore, we need a mechanism using which we can decide when to execute such iterations that access multiple tiles. Our proposed solution to this problem operates as follows. Let us consider the scenario depicted in Figure 4. In this scenario (given for a processor p_k), there is an iteration point (\vec{l}) that belongs to two different iteration sets: $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$ and $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_j))$, corresponding to data tiles $\mathcal{D}(\vec{a}_i)$ and $\mathcal{D}(\vec{a}_j)$, respectively. We also see that $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$ actually accesses both $\mathcal{D}(\vec{a}_i)$ and $\mathcal{D}(\vec{a}_j)$ through two different references, R_i and R_j , respectively. Assuming that $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$ and $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_j))$ are the only sets that contain iteration point \vec{l} , our approach schedules it with $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$ if there is no dependence between these two sets or $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_j))$ depends on $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$. Otherwise, i.e., if $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_i))$ depends on $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_j))$, it schedules \vec{l} with $\mathcal{J}_{p_k}(\mathcal{D}(\vec{a}_j))$. This approach can easily be extended to the case where we have multiple (> 2) iteration sets that contain \vec{l} . In the rest of the paper, when we talk about “scheduling a data tile”, we mean “scheduling the iteration sets (potentially from different processors) that access that data tile”.

Let us illustrate our approach using a simple scenario. Figure 5(a) gives example iteration sets (from 1 to 16) and their data tile access patterns. Note that, as defined earlier, each iteration set accesses a data tile. Assuming that the code is parallelized over two processors, Figure 5(b) gives a possible original schedule order. In this schedule order, intra-processor data reuse is not good, since iteration sets accessing the same data tile for the same processor are not scheduled successively. For example, iteration set 1 and iteration set 5 access the same data tile, but they are scheduled four time slots apart from each other. This schedule is not good in terms of inter-processor data reuse either, since the iteration sets accessing the same data tile are not scheduled together. For example, both iteration set 1 and iteration set 12 access the same data tile, but iteration set 1 is scheduled on P1 at time slot T1 and iteration set 12 is scheduled on P2 at T4. Figure 5(c) gives a schedule order that optimizes for intra-processor data reuse only. In this schedule order, all the iteration sets accessing the same data tile from the same processor are scheduled one after another. However, this schedule is still not optimized as far as inter-processor data reuse is concerned. For example, iteration 1 and iteration 12 are still scheduled at six time slots far away from each other. Figure 5(d) gives a schedule order that optimizes both intra-processor data reuse

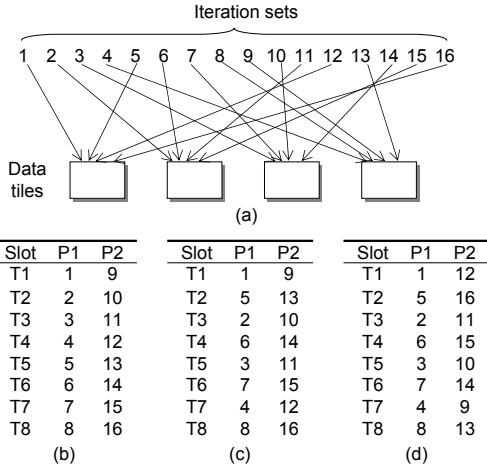


Fig. 5. A simple scenario that illustrates our approach. (a) Data sets accessed by different iteration sets. The numbers represent the iteration sets assigned to processors, and the blocks represent the data tiles. Intra-processor data locality exists if two iteration sets from the same processor accessing the same data tile are scheduled one after another. Inter-processor data locality exists if two iteration sets from two processors accessing the same data tile are scheduled together (i.e., at the same time slot). (b) Original schedule order. T1, T2, ..., T8 represent time slots. (c) A schedule order that optimizes for intra-processor data locality only. (d) A schedule order that optimizes for both intra-processor and inter-processor data locality.

```

initialize  $T_p[]$  to 0 for  $1 \leq p \leq P$ ;
while (exist data tile not scheduled) {
  if (exist data tile shared by two or more processors) {
    if (more than one such data tile) {
      select the one such that it forms convex region with previous scheduled tiles;
      if (such data tile does not exist) { randomly select one; }
    } else select the only data tile;
  } else {
    if (such more than one such data tile) {
      select the one such that it forms convex region with previous scheduled tiles;
      if (such data tile does not exist) { randomly select one; }
    } else select the only data tile;
  }
  assume that the selected one is  $J_p(\mathcal{D}(\vec{a}_i))$ ;
  if the data tile is shared
    find the earliest time slot  $k$  that is available on all the sharing processors;
  else
    find the earliest time slot  $k$  that is available on processor  $p$ ;
   $T_p[k] = i$ ;
}

```

Fig. 6. Compiler algorithm for the dependence-free case. If $T_p[k] == 0$, time slot k has not been assigned to any iteration sets of processor p . If $T_p[k] == n$ ($n \geq 0$), time slot k is assigned to $J_p(\mathcal{D}(\vec{a}_i))$.

and inter-processor data reuse. In this schedule order, determined by our approach, all the iteration sets accessing the same data tile from the same processor are scheduled one after another. In addition, all iteration sets from the different processors accessing the same data tile are scheduled together (i.e., at the same time slot). For example, iteration set 1 and iteration set 12 are both scheduled at T1, on processors P1 and P2 respectively. The objective of our scheduling algorithm is to find a schedule order such that both intra-processor data reuse and inter-processor data reuse are optimized (as in the case of Figure 5(d)). This example also illustrates that optimizing for intra-processor data locality only does not guarantee optimized inter-processor data locality.

B. Scheduling for the Dependence-Free Case

Our compiler algorithm that schedules the iteration sets in a given loop nest for the dependence-free case (i.e., when we do not have any data dependence in the loop nest being optimized) is given in Figure 6. In this algorithm, we give priority to those data tiles shared by two or more processors (i.e., we try to schedule them earlier), since inter-processor reuse is determined based on whether the different processors access these data tiles at the same time. At each step, if there exists any shared data tile, we schedule it at the earliest time slot at which all the sharing processors are available. If there is no shared data tile, then we select one from all the remaining

```

initialize  $T_p[]$  to 0 for  $1 \leq p \leq P$ ;
for (each node in  $\mathcal{G}$ ) {
  if (node has no incoming edge)  $node.st = 0$ ; else  $node.st = \infty$ ;
}
while ( $\mathcal{G} \neq \emptyset$ ) {
  if (exist data tile shared by two or more processors and
  the corresponding nodes have no incoming edge) {
    if (such more than one such data tile) {
      select the one such that it forms convex region with previous scheduled tiles;
      if (such data tile does not exist)
        randomly select one;
    } else
      select the only data tile;
  } else {
    if (more than one such data tile) {
      select the one such that it forms convex region with previous scheduled tiles;
      if (such data tile does not exist)
        randomly select one;
    } else
      select the only data tile;
  }
  Assume that the selected node represents  $J_p^m(\mathcal{D}(\vec{a}_i))$ ;
  if the data tile is shared
    find the earliest time slot  $k$  that available on all the sharing processors;
  else
    find the earliest time slot  $k$  that available on processor  $p$ ;
   $T_p[k].tile = i$ ;  $T_p[k].ss = m$ ;
  for (each othernode that depends on node) {
    if ( $othernode.st < node.st + 1$ )  $othernode.st = node.st + 1$ ;
  }
  remove node from  $calG$ 
}

```

Fig. 7. Compiler algorithm for the data dependence case, assuming that each $J_p(\mathcal{D}(\vec{a}_i))$ has been divided into smaller subsets and \mathcal{G} is the dependence graph. If $T_p[k].tile == 0$, time slot k has not been assigned to any iteration sets of processor p . If $T_p[k].tile == n$ ($n \geq 0$), time slot k is assigned to $J_p^m(\mathcal{D}(\vec{a}_i))$, where $m == T_p[k].ss$. For a node $node$ in \mathcal{G} , $node.st$ gives the earliest time slot that it can be scheduled based on data dependence information.

data tiles, and schedule it. In both the cases, if there are more than one data tiles schedulable, we can either randomly select one, or select the one which serves better as far as minimizing the code generation overheads is concerned. We choose the latter approach in our algorithm. Our criteria is to determine whether a data tile can form a *convex region* with the previously scheduled data tiles. In other words, a data tile that can form convex region with the previously scheduled data tiles has precedence over other data tiles in scheduling. This is because, for each convex region, the code generation tool is able to create a single loop nest that enumerates all the points in this region. Therefore, the number of convex regions determines the number of loop nests required. To reduce the output code size, we need to reduce the number of convex regions in the schedule. When this algorithm finishes, the array $T_p[]$ in Figure 6 gives the time slots at which the iteration sets should be scheduled on processor p .

C. Scheduling for the Case with Data Dependence

Our scheduling strategy has to be slightly modified when we have data dependences. The problem with data dependences in our context is that we may not be able to execute all loop iterations that belong to $J_p(\mathcal{D}(\vec{a}_i))$ for a given $1 \leq i \leq q$. Our solution to this problem is as follows. We focus, without loss of generality, on $J_p(\mathcal{D}(\vec{a}_i))$ for a processor p . We divide $J_p(\mathcal{D}(\vec{a}_i))$ into s subsets: $J_p^1(\mathcal{D}(\vec{a}_i))$, $J_p^2(\mathcal{D}(\vec{a}_i))$, ..., $J_p^s(\mathcal{D}(\vec{a}_i))$. Each subset is small enough so that all the iterations in a subset can be scheduled together under any circumstances. Such a division is always possible since we can have only one iteration in each subset in the extreme case. However, in general, we still want to keep the size of each subset as large as possible to reduce the number of subsets and the associated scheduling and code generation costs. After this division, we can build a *dependence graph* \mathcal{G} . In this graph, each node represents an iteration subset and each edge represents a data dependence from one subset to another. Then, we use a combination of list scheduling and the algorithm given in Figure 6 to schedule the subsets. Our compiler algorithm that schedules the iterations in a given loop nest with data dependences is given in Figure 7. In this algorithm, at each step, we select a schedulable node that has no incoming edges. Similar to the algorithm given in Figure 6, the shared data tiles and the data tiles that can form convex region with the previously scheduled tiles have precedence over other tiles. One point that we need to be careful about is the start time of each node. When there is no dependence, we can schedule an iteration set at the earliest available time slot as presented in Figure 6. When a data dependence exists, however,

a node cannot be scheduled earlier than any node that it depends on. We use *node.st* in Figure 7 to capture the earliest time slot that it can be scheduled based on data dependence information. When a node is scheduled, this information is updated for all the nodes that depends on this node. After all the nodes in the dependence graph are scheduled, the array $T_p[]$ in Figure 7 gives the schedule for each iteration subset of processor p .

D. Extension to Multiple Arrays

Recall that one of the assumptions under which the approach above is presented is that we have only one array. Since in general we can have multiple arrays in the application and each loop nest typically accesses a subset of these arrays, we need to have a more general mechanism than the one described above. We start by observing that what essentially we perform is to re-organize (schedule) loop iterations for each processor based on data tile partitioning of the array. Therefore, when we have multiple arrays, one of the difficulties we have is to select the array using which we can re-organize the loop iterations. However, when we select an array and re-organize loop iterations based on that array, there is no guarantee that the accesses to the other arrays will be good from the data locality perspective. An important observation that can be useful at this point is that the number of arrays accessed by a loop nest is typically not very large. Consequently, if we have a good metric that helps us rank the different selections from the locality angle, (and that can be evaluated at compile time), we can exhaustively try each array as a potential candidate using which we can re-organize loop iterations.

The main challenge before such an approach is the *evaluation metric* to be used. In this work, we use the *number of untimely accesses to shared data* as our metric. Suppose that we have K arrays accessed in a given loop nest and we want to determine the array using which the loop iterations are to be re-organized. Let us focus, without loss of generality, on array X_i where $1 \leq i \leq K$. We assume that this array is divided into q different tiles, namely, $\mathcal{D}(\vec{a}_1)$, $\mathcal{D}(\vec{a}_2)$, $\mathcal{D}(\vec{a}_3)$, \dots , $\mathcal{D}(\vec{a}_q)$. Now, let us focus on a specific tile $\mathcal{D}(\vec{a}_j)$. Using polyhedral algebra (i.e., using the Omega Library), we can identify the set of iterations, among those assigned to processor p as a result of parallelization, that access the elements in this tile, that is, the set $\mathcal{J}_p(\mathcal{D}(\vec{a}_j))$. We can now determine the set of data elements accessed by the iterations in $\mathcal{J}_p(\mathcal{D}(\vec{a}_j))$ from arrays other than X_i (i.e., $\forall k: X_k, k \neq i$). We denote these sets as:

$$\mathcal{D}'_{p,X_1}(\vec{a}_j), \mathcal{D}'_{p,X_2}(\vec{a}_j), \dots, \mathcal{D}'_{p,X_{i-1}}(\vec{a}_j), \mathcal{D}'_{p,X_{i+1}}(\vec{a}_j), \dots, \mathcal{D}'_{p,X_K}(\vec{a}_j).$$

Note that $\mathcal{D}'_{p,X_k}(\vec{a}_j)$ captures the set of elements accessed by processor p from array X_k as a result of executing iterations in set $\mathcal{J}_p(\mathcal{D}(\vec{a}_j))$. We can write similar sets for other processors as well. That is, we can list the sets:

$$\begin{aligned} &\mathcal{D}'_{1,X_1}(\vec{a}_j), \mathcal{D}'_{1,X_2}(\vec{a}_j), \dots, \mathcal{D}'_{1,X_{i-1}}(\vec{a}_j), \mathcal{D}'_{1,X_{i+1}}(\vec{a}_j), \dots, \mathcal{D}'_{1,X_K}(\vec{a}_j) \\ &\mathcal{D}'_{2,X_1}(\vec{a}_j), \mathcal{D}'_{2,X_2}(\vec{a}_j), \dots, \mathcal{D}'_{2,X_{i-1}}(\vec{a}_j), \mathcal{D}'_{2,X_{i+1}}(\vec{a}_j), \dots, \mathcal{D}'_{2,X_K}(\vec{a}_j) \\ &\dots \\ &\mathcal{D}'_{p,X_1}(\vec{a}_j), \mathcal{D}'_{p,X_2}(\vec{a}_j), \dots, \mathcal{D}'_{p,X_{i-1}}(\vec{a}_j), \mathcal{D}'_{p,X_{i+1}}(\vec{a}_j), \dots, \mathcal{D}'_{p,X_K}(\vec{a}_j) \end{aligned}$$

Now, let us discuss set the following expression:

$$\Delta(X_i) = \sum_{j=1}^q \sum_{k=1, k \neq i}^K \sum_{s=1}^P \sum_{s_2=1, s_2 \neq s_1}^P |\mathcal{D}'_{s_1, X_k}(\vec{a}_j) \cap \mathcal{D}'_{s_2, X_k}(\vec{a}_j)|$$

This expression gives us the total number of elements accessed by processor pairs from all arrays except X_i when all tiles of array X_i are considered. Ideally, we want the value of $\Delta(X_i)$ to be as large as possible (which lead to maximum amount of data reuse). Our approach selects array X_e as the one to be used for re-organizing loop iterations if $\Delta(X_e) \geq \Delta(X_i), \forall i \neq e$.

III. RELATED WORK

Memory hierarchy is a critical component in determining the system performance. There are various prior efforts [17], [10], [16], [9], [1], [5] that try to improve data locality using high-level code restructuring techniques. It has been shown that linear loop transformation is an effective technique for improving data locality in the context of array/loop-intensive applications [17], [10]. Non-linear loop transformations, such as loop tiling, can also be used to improve

data locality [16]. Kodukula and Pingali proposed a data-centric loop transformation framework targeting at general loop structures [9]. Their work is similar to ours in that they try to bring the computations working on the same data set together to improve data locality. But, they focus on only intra-processor data reuse. In comparison, we are interested in improving inter-processor data reuse as well as intra-processor data reuse.

MPSoC architectures are becoming popular for designing embedded systems. They gain ground in both academic environments and industry [4], [11], [12], [14]. Hammond et al [7] compared three alternative microarchitectures: MPSoC, SMT, and superscalar. They found both software and hardware trends favor the MPSoC architecture over the others. There exist several prior studies focusing on the different aspects of MPSoC, for example, memory performance, communication, reliability, etc [6], [13], [15], [18]. Our work complements the previous work, and focuses on a new aspect of MPSoC, that is, inter-processor data locality.

IV. CONCLUSION

Recent research indicates that packing multiple processor cores on the same die is an effective way of utilizing the increasing number of transistors. One of the main advantages of placing multiple cores into a single die is that it helps reduce the on-chip communication costs between the processor cores that are typically very high in conventional high-performance parallel architectures. However, on the negative side, this tighter integration exerts an even higher pressure on off-chip accesses to the memory system. This makes minimizing the number of off-chip accesses a critical optimization goal. This paper discusses a compiler-based solution to this problem. The idea behind the proposed approach is to reduce the reuse distance for data accessed by more than one processor. Specifically, the proposed approach optimizes inter-processor data reuse by re-organizing loop iterations of each processor carefully, considering how data elements are shared across processors.

ACKNOWLEDGEMENT

This work is supported in part by NSF Career Award 0093082 and a grant from GSRC.

REFERENCES

- [1] F. Cathoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.
- [2] Omega library. <http://www.cs.umd.edu/projects/omega>.
- [3] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, 1993.
- [4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *Proc. of ISCA*, 2000.
- [5] S. Carr, K. S. McKinley, and C. Tseng. Compiler Optimizations for Improving Data Locality. *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.
- [6] M. Gomaa, C. Scarborough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proc. International Symposium on Computer Architecture*, 2003.
- [7] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer Special Issue on "Billion-Transistor Processors"*, September 1997.
- [8] I. Kadayif, M. Kandemir, and U. Sezer. An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors. In *Proc. of Design Automation Conference*, New Orleans, LA, June 2002.
- [9] I. Kodukula and K. Pingali. Data-Centric Transformations for Locality Enhancement. *International Journal of Parallel Programming*, October 2001.
- [10] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, Yale University, August 1992.
- [11] MAJC-5200. <http://www.sun.com/microelectronics/MAJC/5200wp.html>
- [12] MP98: A Mobile Processor. <http://www.labs.nec.co.jp/MP98/top-e.htm>.
- [13] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. ISCA*, 1994.
- [14] POWER4 System Microarchitecture, *White Paper*, <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>
- [15] S. Richardson. MPOC: A chip multiprocessor for embedded systems. Technical Report HPL-2002-186, HP Labs, 2002.
- [16] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Proc. of the Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987.
- [17] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. PLDI*, June 1991.
- [18] W. Wolf. The future of multiprocessor systems-on-chips. In *Proc. Design Automation Conference*, 2004.