Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors

Qingda Lu
The Ohio State University
luq@cse.ohio-state.edu

Christophe Alias
ENS Lyon – INRIA, France
Christophe.Alias@ens-lyon.fr

Uday Bondhugula, Thomas Henretty
The Ohio State University
{bondhugu,henretty}@cse.ohio-state.edu

Sriram Krishnamoorthy
Pacific Northwest National Lab
sriram@pnl.gov

J. Ramanujam
Louisiana State University
jxr@ece.lsu.edu

Atanas Rountev, P. Sadayappan
The Ohio State University
{rountev,saday}@cse.ohio-state.edu

Yongjian Chen
Intel Corp.
yongjian.chen@intel.com

Haibo Lin
IBM China Research Lab
linhb@cn.ibm.com

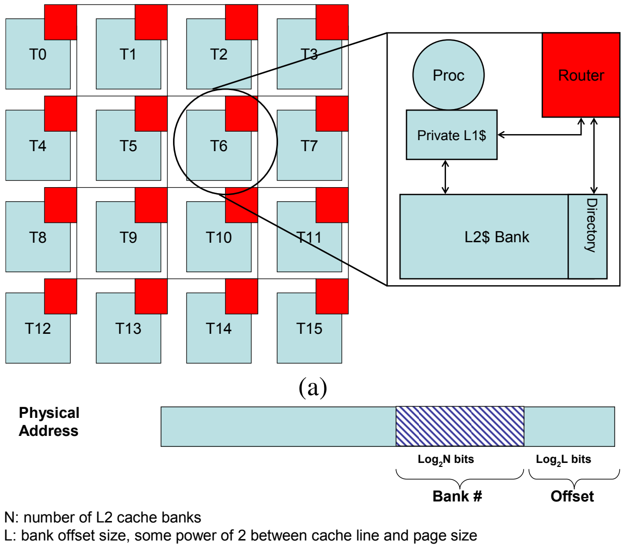
Tin-Fook Ngai
Intel Corp.
tin-fook.ngai@intel.com

***Abstract*—With increasing numbers of cores, future CMPs (Chip Multi-Processors) are likely to have a tiled architecture with a portion of shared L2 cache on each tile and a bank-interleaved distribution of the address space. Although such an organization is effective for avoiding access hot-spots, it can cause a significant number of non-local L2 accesses for many commonly occurring regular data access patterns. In this paper we develop a compile-time framework for data locality optimization via data layout transformation. Using a polyhedral model, the program's localizability is determined by analysis of its index set and array reference functions, followed by non-canonical data layout transformation to reduce non-local accesses for localizable computations. Simulation-based results on a 16-core 2D tiled CMP demonstrate the effectiveness of the approach. The developed program transformation technique is also useful in several other data layout transformation contexts.**

***Keywords*-Data Layout Optimization, Polyhedral Model, NUCA Cache**

I. Introduction

Many proposed chip multiprocessor (CMP) designs [1], [2] feature shared on-chip cache(s) to reduce the number of off-chip accesses and simplify coherence protocols. With diminutive feature sizes making wire delay a critical bottleneck in achieving high performance, proposed shared caches employ a Non-Uniform Cache Architecture (NUCA) design that spreads data across cache banks that are connected through an on-chip network [3], [4]. Fig. 1(a) shows a tiled architecture for a CMP. Each tile contains a processor core with private L1 cache, and one bank of the shared L2 cache. Future CMPs are likely to be based on a similar design to enable large numbers of processors and large on-chip caches to be connected through emerging technologies such as 3D stacking. Fig. 1(b) shows the mapping of a physical memory address to banks in the L2 cache. The address space is block-cyclically mapped across the banks with a block size L. The lowest $log_2 L$ bits represent the displacement within a block mapped to a bank, and the next set of $log_2 P$ bits specify the bank number. The value of L depends on the granularity of interleaving across the banks, which can range from one cache line to an OS page. Cache line interleaving has a number of benefits, including uniform distribution of arrays across the banks, enabling high collective bandwidth in accessing its elements and minimization of hot-spots. Since bank interleaving at cache line granularity has been used in most NUCA CMP designs, the evaluations in this paper assume cache-line interleaving. However the developed approach is also applicable to coarser interleaving across banks.



(b)

Figure 1. (a) Tiled CMP architecture. (b) Mapping a physical address to a cache bank on a tiled CMP.

In this paper, we develop a compile-time data locality optimization framework for bank-interleaved shared cache systems. We first use a simple example to illustrate the optimization issue and the proposed data layout transformation approach.

```
    while (condition) {
      for (i = 1; i < N-1; i++)
        B[i] = A[i-1] + A[i] + A[i+1]; //stmt S1
      for (i = 1; i < N-1; i++)
        A[i] = B[i];  //stmt S2 }
```

(a) Sequential 1D Jacobi code.

```
Compute(int threadID) {
  processor_bind(threadID);
  NB = ceiling(N / (P * L)) * L;
  while (condition) {
    for (i = threadID*NB; i < min(N,(threadID+1)*NB);i++){
      B[index(i)] = A[index(i-1)] + A[index(i)]
                  + A[index(i+1)];}
    Barrier();
    for (i = threadID*NB; i < min(N,(threadID+1)*NB);i++){
      A[index(i)] = B[index(i)]; }
    Barrier();}
}
```

(b) Parallel 1D Jacobi code with non-canonical data layouts.

Figure 2. 1D Jacobi code.

Consider the code in Fig. 2(a), for 1-D Jacobi iteration. In statement S1 of the code, iteration i accesses $A[i-1]$, $A[i]$, and $A[i+1]$. Completely localized bank access is impossible with this code for any load-balanced mapping of iterations to processors. For ease of explanation, we assume that the array size N is a large power of two. Assuming there are four processors on a tiled CMP and $L = 2$ is the cache line size in elements, there are many ways of partitioning the iteration space to parallelize this program. Consider the following two mapping schemes with one thread on each processor: (1) based on the *owner-computes* rule, each thread is responsible for computing the data elements mapped to its local bank — this corresponds to OpenMP's static scheduling with a chunk size of L; (2) the iteration space is divided evenly into four contiguous partitions and they are assigned to the processors in order — this corresponds to OpenMP's default static scheduling. Note that aligning the base addresses of A and B is important since A[i] and B[i] are always referenced together. Assume that the origins of A and B are both aligned to bank 0's boundary. Fig. 3(a) shows the data communication needed between different processors when the computation is partitioned using mapping (1). In order to compute B's elements in a cache line, the corresponding cache line of A, as well as the previous and next cache line are required. (for the A[i-1] reference for the lowest iteration in the block and A[i+1] reference for the highest iteration in the block). With two cache lines of A being remote, the total inter-bank communication volume in every outer iteration is roughly $2N$. For mapping (2), if we keep the original layout of each array, most data accesses by each processor are remote. Furthermore, such remote communications can be across the entire chip instead of only among neighboring tiles. However, it is possible to rearrange the data layout in order to significantly enhance locality of access and maintain affinity between computation and data. This is shown in Fig. 3(b). As illustrated, there are only six remote accesses in every outer iteration. Fig. 2(b) shows an outline of the threaded parallel code with such non-canonical data layouts. A non-linear indexing function is used to map each array index to its memory location. Index calculation with such a function can be highly optimized as discussed in Sec. IV. The code also binds each thread to its corresponding processor to maintain processor / data affinity.

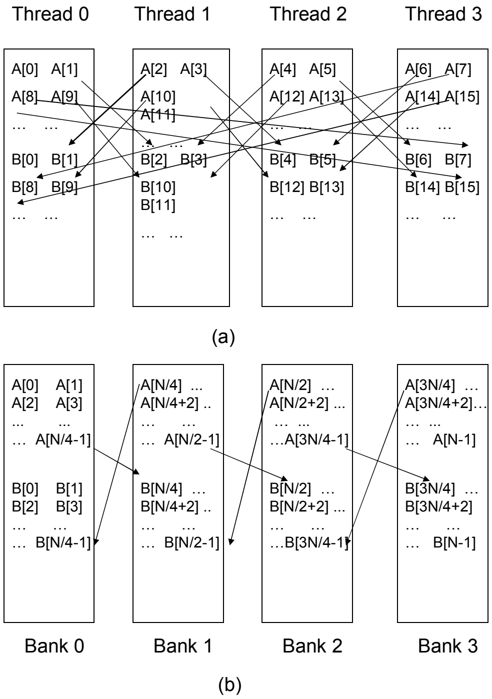


Figure 3. Parallelizing a 1D Jacobi program on a 4-tile CMP.

The basic problem illustrated by this example is a mismatch between the fundamental data affinity relationships inherent to the algorithm and the locality characteristics imposed by the hardware bank-mapping. The nature of the Jacobi computation is that each data element of the 1D array A has affinity with its immediate neighbors, since they are operands in common arithmetic operations. Thus data locality can be optimized by grouping a set of contiguous elements of A on a processor, along with mapping the operations that access the elements. However, this is at odds with the block-cyclic bank mapping imposed by the hardware which maps successive cache lines to distinct banks in order to minimize hot spots. The solution is to perform a suitable data layout transformation so that data elements with strong affinity to each other get mapped to the same processor's bank under the imposed hardware mapping scheme.

The paper develops a general framework for integrated data layout transformation and loop transformations for enhancing data locality on NUCA CMP's. While the direct focus of the paper's presentation is to detail the compilation framework for this target, the basic approach has much broader applicability that is the subject of ongoing work - effective vectorization for short SIMD architectures, domainspecific language and compiler for stencil computations, and a compiler for linear algebra with recursive arrays [5].

Our framework uses the polyhedral model of compilation. The polyhedral model provides a powerful abstraction to reason about transformations on collections of loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space called the statement's *polyhedron*. With the polyhedral model, it is possible to reason about the correctness of complex loop transformations using powerful machinery from linear algebra and integer linear programming. A significant advantage of using a polyhedral compiler framework is that imperfectly nested loops are very cleanly handled as are compositions of loop and data transforms. Although the foundational ideas for polyhedral compilation were developed in the early nineties [6], [7], [8], it was generally considered to be too expensive for practical use. However a number of recent developments [9], [10], [11], [12] have demonstrated the practical effectiveness of the approach. Now at least two commercial compilers are known to incorporate polyhedral analyses; the latest release of GCC [13] also includes it.

The approach we develop has two main steps. First, an affinity characterization of the data arrays is carried out using a "localization" analysis. In this step an affine mapping of iteration spaces and data spaces onto a logical 1D target space is sought such that no iteration accesses any data that is beyond a bounded distance in the target space. If such an affine mapping exists, our analysis is guaranteed to find it. The existence of such a mapping implies that the computation is localizable, i.e. data affinities among the elements of arrays are constrained to be close by along some dimension of the array. Consequently, a mapping of iterations to processors along with a suitably transformed data layout is feasible that will result in enhanced data locality.

The localization analysis identifies dimensions of data arrays that are to be layout-transformed. A key issue is that of efficient code generation that indexes into the transformed arrays. This is done by using CLooG [10]. A straightforward use of CLooG results in very inefficient code with significant indexing overhead in the innermost loops. Efficient code is generated by differentiating the "bulk" scenario from the "boundary" case via linear constraints; the former region is traversed without expensive indexing calculations, which are only needed for the smaller boundary regions.

The paper makes the following contributions:

- It develops an approach for automatic data layout transformation to enhance data locality on emerging NUCA architectures.
- It develops an effective code generation framework that can be adapted for an important class of data layout transformations, including those to enhance vectorization of stencil codes.

The rest of the paper is organized as follows. We first present the framework that characterizes data affinity via data localization analysis in Sec. II. In Sec. III, we describe the affinity characterization is used to determine data layout transformations. The key challenge of efficient code generation to access the layout-transformed data is addressed in Sec. IV. In Sec. V, using several benchmarks we evaluate the effectiveness of our approach with an architecture simulator We discuss related work in Sec. VI and present our conclusions in Sec. VII.

II. Localization Analysis

This section describes the procedure for localization analysis. We begin by providing some background, using notation similar to that used by Griebl [14].

A. Polyhedral Framework for Program Transformation

A d-dimensional function f of n variables $x_1, x_2, \ldots, x_n$ is *affine* if and only if $f(\vec{x}) = M_f\vec{x} + c$, where $\vec{x} = \begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}^T$, $M_f \in \mathbb{R}^{d\times n}$ and $c \in \mathbb{R}^{d\times 1}$. An affine function f can be expressed in a linear form by introducing a special constant parameter "1": $f(\vec{x'}) = M'_f\vec{x'}$, where $\vec{x'} = \begin{bmatrix} x_1 & \ldots & x_n & 1 \end{bmatrix}^T$ and $M'_f = [M_f \;\; c]$.

A *hyperplane* is a $(d-1)$-dimensional affine subspace of a d-dimensional space and can be represented by an affine equality $Mx+c = 0$. A *halfspace* consists of all points of the d-dimensional space which are on one side of a hyperplane, including the hyperplane. A halfspace can be represented by an affine inequality $Mx + c \geq 0$. A *polyhedron* is the intersection of a finite number of halfspaces. A *polytope* is a bounded polyhedron.

The bounds of the loops surrounding a statement S are expressed as a system of linear inequalities that form a polytope, represented as $D_S(\vec{i}) \geq 0$, where $\vec{i}$ includes the iteration vector and all structure parameters. An array access in statement S to array A is denoted $\mathbf{F}_{S,A}(\vec{i})$. Data dependences are described using *h-transformations*. An h-transformation is an affine function that takes a destination statement instance and gives the source statement instance [6]. Note that we are using value-based dependences. The domain of the function is denoted explicitly as a polytope $P_e(\vec{i}) \geq 0$.

We are interested in a *computation allocation* π_S that is an affine function mapping every dynamic instance of a statement S to an integer vector that represents a virtual processor. π_S must satisfy the constraints imposed by loop bounds and data dependences. Similar to computation allocation, a *data mapping* ψ_A maps every array element to a virtual processor.

The problems of finding π_S and ψ_A are often formalized as optimization problems. However, such optimization problems are not affine because coefficients in π_S or ψ_A and the iteration vector are all unknowns. *Farkas Lemma* [6] is used in such cases for quantifier elimination. This lemma states how such a system of affine inequalities with quantifiers can be transformed to a system of affine equalities by adding non-negative variables, referred to as Farkas multipliers. After such transformations π_S and ψ_A can be determined by using a linear programming solver.

B. Localization Constraints

We define a program as *localizable* if there exists a computation mapping and a data mapping to a target 1-dimensional space such that the data required by any iteration instance is mapped in a bounded neighborhood in the target space. The existence of such a mapping enables suitable layout transformation to enhance locality and reduce the total amount of non-local data accesses. The localizability constraints for computation allocation are formalized as follows.

Definition 1: (Localized computation allocation / data mapping) For a program P, let D be its index set, computation allocation π and data mapping ψ for P are localized if and only if for any array A, and any reference $\mathbf{F}_{S,A}$, $\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies |\pi_S(\vec{i}) - \psi_A(\mathbf{F}_{S,A}(\vec{i}))| \leq q$, where q is a constant.

As a special case, *communication-free localization* can be achieved if and only if for any array A and any array reference $\mathbf{F}_{S,A}$ in a statement S, computation allocation π and data mapping ψ satisfy $\psi_A(\mathbf{F}_{S,A}(\vec{i})) = \pi_S(\vec{i})$.

C. Localization Analysis Algorithm

The localization analysis based on the above discussion consists of the following steps, as described in Algorithm 1.

Step 1: Grouping Interrelated Statements/Arrays We determine connected sets of statements in an input program. We form a bipartite graph where each vertex corresponds to a statement or an array, and edges connect each statement vertex to all arrays referenced in that statement. We then find the connected components in the bipartite graph. The statements in each connected component form an equivalence class.

Step 2: Finding Localized Computation Mapping Following Def. 1, we formulate the problem as finding an affine computation allocation π and an affine data mapping ψ that satisfy $|\pi_S(\vec{i}) - \psi_A(\mathbf{F}_{S,A}(\vec{i}))| \leq q$ for every array reference $\mathbf{F}_{S,A}$. The π function identifies a parallel space dimension in the iteration space of each statement. Therefore a system of such inequalities is formed as constraints for all statements and array references.

Note that the equation in Def. 1 is not affine due to the quantifier. We need to first rewrite each constraint as

$$\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies \pi_S(\vec{i}) - \psi_A(\mathbf{F}_{S,A}(\vec{i})) + q \geq 0; \qquad (1)$$
$$\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies -\pi_S(\vec{i}) + \psi_A(\mathbf{F}_{S,A}(\vec{i})) + q \geq 0, \quad (2)$$

For simplicity of presentation, we refer to $\pi_S(\vec{i}) - \psi_A(\mathbf{F}_{S,A}(\vec{i})) + q$ and $-\pi_S(\vec{i}) + \psi_A(\mathbf{F}_{S,A}(\vec{i})) + q$ as $f_1(\vec{i})$ and $f_2(\vec{i})$ respectively.

We apply Farkas Lemma to transform the above constraints to affine equalities by introducing Farkas multipliers. Take the equation in (1) as an example. With Farkas Lemma, we have $\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies f_1(\vec{i}) \geq 0$ if and only if $f_1(\vec{i}) \equiv \lambda_0 + \sum_k \lambda_k(a_k\vec{i} + b_k)$ with $\lambda_k \geq 0$, where affine inequalities $a_k\vec{i} + b_k \geq 0$ define D_S, and λ_k are the Farkas multipliers. Therefore we have the following:

$$M'_{f_1}\begin{bmatrix}\vec{i}\\1\end{bmatrix} = \begin{bmatrix}\lambda_1 & \dots & \lambda_m & \lambda_0\end{bmatrix}\begin{bmatrix}M'_{D_S}\\0\dots01\end{bmatrix}\begin{bmatrix}\vec{i}\\1\end{bmatrix}, \vec{\lambda} \geq 0 \qquad (3)$$

where $f_1(\vec{i}) = M'_{f_1}\begin{bmatrix}\vec{i}\\1\end{bmatrix}$ and $D_S(\vec{i}) = M'_{D_S}\begin{bmatrix}\vec{i}\\1\end{bmatrix}$. Since Eq.(3) holds for all $\vec{i}$,

$$M'_{f_1} = \begin{bmatrix}\lambda_1 & \dots & \lambda_m & \lambda_0\end{bmatrix}\begin{bmatrix}M'_{D_S}\\0\ \dots\ 0\ 1\end{bmatrix}, \vec{\lambda} \geq 0 \quad (4)$$

We apply Fourier-Motzkin elimination to Eq. (4) to remove all the Farkas multipliers and obtain a new set of inequalities. The whole procedure is also applied to the equation in (2) to obtain another set of inequalities. These two sets together form the inequalities that constrain the coefficients of π and ψ.

In order to minimize potential communication, the system of generated inequalities is solved as an integer programming problem. $min(q)$ is used as the objective function to minimize the communication overhead. If such a q is identified, we find localized computation allocation and data mapping. As a special case, if q is determined to be 0, the computation is identified to have communication-free computation localization. If a non-zero q is determined as the minimal solution, the program is not communication-free but is localizable. If a solution is not obtained by the above steps, the program is identified as non-localizable.

Algorithm 1 Localization analysis algorithm

```
Require: Array access functions after indices are rewritten to access byte arrays
$C = \emptyset$
for each array reference $\mathbf{F}_{S,A}$ do
    Obtain new constraints: $\pi_S(\vec{i}) - \psi_A(\mathbf{F}_{S,A}(\vec{i})) + q \geq 0$ and $\psi_A(\mathbf{F}_{S,A}(\vec{i})) - \pi_S(\vec{i}) + q \geq 0$ under $\vec{i} \in D_S$.
    Apply Farkas Lemma to new constraints to obtain linear constraints; eliminate all Farkas multipliers
    Add linear inequalities from the previous step into $C$
end for
Add objective function $(\min q)$
Solve the resulting linear programming problem with constraints in $C$
if $\psi$ and $\pi$ are found then
    return $\pi$, $\psi$, and $q$
end if
return "not localizable"
```

Partial Localization We also consider partial localization. If localized π and ψ cannot be found, we attempt to find partial localization by removing constraints on small arrays with high reuse. Then we recompute π and ψ, seeking localization with the reduced constraints — localization of small arrays being ignored in the expectation that temporal locality may be exploited via registers or in the private L1 caches.

As an example of partial localization, code to compute matrix-vector multiplication is shown in Fig. 4. There are two sets of data dependences between instances (i, j) and (i', j') of statement 2: (1) output dependence on array C:

```
for (i = 0; i < N; i++) {
  C[i] = 0;  //stmt 1
  for (j = 0; j < N; j++)
    C[i] = C[i]+A[i][j]*B[j] //stmt 2 }
```

Figure 4. Matrix-vector multiplication code.

$i - i' = 0, j - j' \geq 1$, and (2) input dependence on array B: $i - i' \geq 1, j - j' = 0$. No localizable mapping exists for all three arrays when considering both dependences. To overcome this problem we remove constraints on array B because B's reuse is asymptotically higher than A's and its size is asymptotically smaller. We are then able to obtain $\pi_1(i,j) = i$, $\pi_2(i,j) = i$, $\psi_A(i,j) = i$ and $\psi_C(i) = i$ that are communication-free regarding arrays A and C.

III. Data Layout Transformation

In this section, we develop the approach to data layout transformation for arrays identified by the localization analysis framework.

As discussed in the previous section, in our implementation the linear programming solver finds a lexicographically minimum solution. We consider data mappings of the form $\psi_A(\vec{d}) = s_A \ d_k + \mathit{off}_A$, where s_A and off_A are small integers, specifying A's distribution stride and offset along its kth dimension. While our localization analysis framework is general and our code generation framework can be extended to handle more complicated programs such as those having skewed data mapping $\psi(d_1, d_2) = d_1 + d_2$, in the rest of the paper we focus on the above common cases. For these programs, each array not removed by partial localization has exactly one data dimension that corresponds to a surrounding parallel loop in the generated code. When such a program is determined to be localizable, we decide each array's data layout to minimize on-chip communication as follows.

In a connected component of a bipartite graph representing a program, if we find communication-free localization where all arrays share the same stride along their fastest varying dimensions, we use the canonical data layout for each array as the computation can be arranged such that each processor core only accesses local data. Otherwise we employ non-canonical data layouts to avoid on-chip communication based on the following analysis. If a program is determined to be localizable but not communication-free, we obtain a small positive q that indicates the maximum difference in the target space, of the affine mappings of any iteration and any data element that it accesses. This means that the maximum distance of separation is $2 * s$ along d_k for any pair of array elements that are accessed by any common statement iteration instance. This implies that by partitioning the array elements into P contiguous groups along d_k and by similarly grouping the corresponding accessor iterations, excellent data locality could be achieved if the array were block-distributed along d_k among the P processors. But the NUCA architecture we assume uses a round-robin bank-mapping of cache lines. Since consecutive cache lines are mapped to different banks, in order to have two cache lines of consecutive data mapped to the same bank, we need to keep them at a distance of PL in virtual memory, where L is the cache line size and P is the number of processor cores. The 1D Jacobi code in Fig. 2 demonstrates such an example. The program is localizable but not communication-free with $q = 1$, $\pi(i) = i$ and $\psi(i) = i$. As shown in 3, the non-canonical data layout mentioned above eliminates most on-chip communication. When the program is communication-free but array mappings do not shared the same stride, we can follow the same approach to avoiding on-chip communication with the non-canonical layout.

The non-canonical layout discussed above can be viewed as the combination of a set of data layout transformations, including *strip-mining*, *permutation* and *padding*. Similar layout transformations have been used by previous studies [15], [16], [17] to avoid conflict misses or reduce false sharing. However, we are unaware of any prior work that has developed a general source-to-source code transformation algorithm to automatically convert the original code addressing canonical arrays into efficient code indexing layout-transformed arrays. In the next section, we develop such a framework.

Strip-mining Strip-mining is analogous to loop tiling in the iteration space. After strip-mining, an array dimension N is divided into two virtual dimensions $(\lceil \frac{N}{d} \rceil, d)$ and an array reference [...][i][...] becomes [...][$\frac{i}{d}$][$i \bmod d$][...]. Note that strip-mining does not change an array's actual layout but instead creates a different logical view of the array with increased dimensionality. As mentioned above, on a tiled CMP, a canonical view does not provide any information about an array element's placement. By strip-mining the fastest varying dimension we have an additional dimension representing data placement. For example, assuming that a one-dimensional array $A(N)$ with canonical layout is aligned to the boundary of bank 0, after strip-mining this array twice, there is a three-dimensional array $A'(\lceil \frac{N}{PL} \rceil, P, L)$ where L is the L2 cache line size and P is the number of tiles. With this three-dimensional view, we can decide an array element's home tile by examining its second fastest varying dimension.

Permutation Array permutation is analogous to loop permutation in the iteration space. After permutation, an array $A(..., N_1, ..., N_2, ...)$ is transformed to $A'(..., N_2, ..., N_1, ...)$ and an array reference $A[...][i_1][...][i_2][...]$ becomes $A'[...][i_2][...][i_1][...]$. Combined with strip-mining, permutation changes an array's actual layout in the memory space. Fig. 5 shows how one-dimensional array A in the 1D Jacobi example in Fig. 2 is transformed with strip-mining and permutation to group consecutive data, where consecutive caches in the array are separate at a distance of PL in the memory. Fig. 5(a) shows the default data layout through strip-mining, where an element $A[i]$ is mapped to tile $i_p = \lfloor i/L \rfloor \bmod P$. In order to realize consecutive data grouping, we can first strip-mine array A(N) so we have $A'(\lceil \frac{N}{PL} \rceil, P, L)$ and then we have array reference $A[i]$ rewritten as $A'[i_p][i_c][i_L]$. As shown in Fig. 5(b), by permuting the first two dimensions of A', we obtain a new array A'' and we are able to map

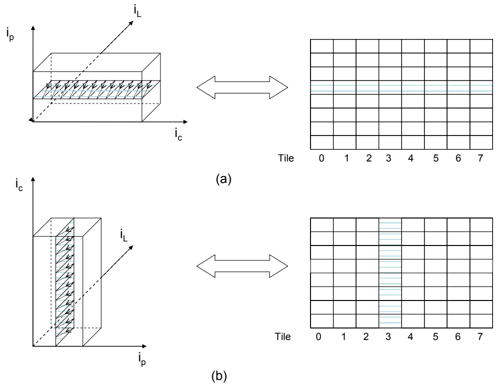


Figure 5. A one-dimensional example of data transformations. (a) Canonical layout after strip-mining. (b) Non-canonical layout after strip-mining and permutation.

$A'[i_p][i_c][i_L]$(i.e., $A''[i_c][i_p][i_L]$) to tile i_p such that block distribution of A is achieved.

Padding To enforce data mapping, array padding is employed in two different ways. First, keeping the base addresses of arrays aligned to a tile specified by the data mapping function can be viewed as *inter-array padding*. In our implementation, we use compiler directives such as __attribute__ in GNU C to control an array's base address. Second, *intra-array padding* aligns elements inside an array with "holes" in order to make a strip-mined dimension divisible by its sub-dimensions. In the 1D Jacobi example, if N is not a multiple of PL in Fig. 5, we pad N to $N' = \lceil \frac{N}{PL} \rceil \cdot P \cdot L$ and $N' - N$ elements are unused in the computation.

In summary, we choose the data layout for each array in a localizable program and map array indices as follows.

- If we find communication-free localization where all arrays share the same stride along their fastest varying dimensions, canonical data layout is employed.
- Otherwise we follow localization analysis results to create a blocked view of n-dimensional array A along dimension k. When $k = 1$ which is the fastest-varying dimension, necessary data layout transformations have been discussed in the above 1D Jacobi example. We have a function σ that maps the original array index to the index with the non-canonical layout. Here $\sigma(i_n, i_{n-1}, ..., i_2, i_1) = (i_n, i_{n-1}, ..., i_2, i_1 \bmod (N_1/P))/L, i_1/(N_1/P), i_1 \bmod L)$. When $k > 1$, the distributed dimension and the fastest varying dimension are strip-mined to add processor and cache line offset dimensions respectively. Padding is needed if $P \nmid N_k$ or $L \nmid N_1$. Finally an index permutation is applied to the processor and the second fastest-varying dimensions. We have $\sigma(i_n, ..., i_k, ..., i_1) = (i_n, ..., i_1/L, i_k \bmod (N_k/P) ..., i_2, i_k/(N_k/P), i_1 \bmod L)$.

IV. Code Generation

As discussed above, data layout transformation on arrays can be expressed as a one-to-one mapping σ from the current n-dimensional index domain to the transformed $(n+2)$-dimensional index domain. A straightforward approach to code generation would be to substitute every array reference $A[u(i))]$ with $A'[\sigma(u(i))]$ where A' is the transformed array, and to generate the code satisfying the given schedule by using the state-of-the-art code generation methods [9]. Unfortunately such a method would produce very inefficient programs as σ involves expensive expressions with integer divisions and modulos that are computed for every array reference.

The method proposed here is to iterate directly in the data space of the transformed array, so that array references are addressed directly by the iteration vector and therefore no longer involve expensive integer expressions. The usual way to generate code in the polyhedral model is to specify for each assignment an iteration domain D and an (affine) scheduling function that gives the execution date for each iteration i in D. Several approaches have been devoted to generate efficient code including the Quilleré-Rajopadhye-Wilde algorithm [9] which is implemented in the state-of-the-art tool CLooG [10]. The approach described hereafter follows this general methodology.

Given an assignment with uniform references $B[u_0(i)] = f(A[u_1(i)], ..., A[u_n(i)])$, whose iteration domain is D, one can write a new domain $D' = \{(i, j_0, ..., j_n), i \in D, j_0 = \sigma(u_0(i)), \ldots, j_n = \sigma(u_n(i))\}$, and generate the code for the schedule provided by the previous step, expressed in terms of a given j_ℓ.

Even though σ involves non-affine expressions with integer division and modulos, it is possible to express each constraint $j_k = \sigma(u_k(i))$ with affine constraints by using the definition of integer division. However, this method can generate a significant number of intermediate variables, leading to a code blow-up. We avoid this problem by using the fact that the inverse of σ is (here) always affine, and to replace each constraint $j_k = \sigma(u_k(i))$ by $\sigma^{-1}(j_k) = u_k(i), j_k \in \mathcal{T}(\sigma)$ where $\mathcal{T}(\sigma)$ is the target domain of σ. This way, no intermediate variables will be needed to describe D'.

For 1-D Jacobi, original code for the first statement is: $B[i] = A[i-1] + A[i] + A[i+1]$. With data layout transformation $\sigma(i) = (i', p, l)$, the transformed code is given by: $B'[i'][p][l] = A'[i'_-][p_-][l_-] + A'[i'][p][l] + A'[i'_+][p_+][l_+]$ where A' and B' are the transformed versions of A and B, respectively. In addition, here $\sigma(i-1) = (i'_-, p_-, l_-)$ and $\sigma(i+1) = (i'_+, p_+, l_+)$.

If we compute σ, one would generate a loop nest with counters t, p, i' and l containing assignments defining the remaining variables in terms of (i', p, l) (j_ℓ), asserting that $(i'_-, p_-, l_-) = \sigma(\sigma^{-1}(i', p, l) - 1)$ and $(i'_+, p_+, l_+) = \sigma(\sigma^{-1}(i', p, l) + 1)$. These expressions involve integer divisions and modulos and must be computed at every iteration, leading to an expensive code.

To bypass this issue, we identify an affine relation between the (transformed) array references. As we apply data layout transformations which are always simple affine expressions followed by an integer division, or a modulo by a constant, this can be achieved by using simple recipes on integer division and modulo. Indeed, when $u << N$, one can remark that $(i-u)/N$ is often equal to i/N, and that $(i-u) \bmod N$ is often equal to $i \bmod N - u$. Applying these recipes to every $j_k = \sigma(u_k(i))$, we obtain a collection of affine relations ($j_k = \alpha_k j_\ell + \beta_k, 0 \le k \le n$) that characterize a class of execution instances of the assignment whose generic instance is drastically simplified, $B'[\alpha_0.j_\ell + \beta_0] = f(A'[\alpha_1.j_\ell + \beta_1], ..., A'[j_\ell], ..., A'[\alpha_n.j_\ell + \beta_n])$, and corresponds to a form of steady state of the computation in the iterations space.

Assume that we have computed $\sigma(i) = (i', p, l)$ using integer division and modulo operations. Using the observation mentioned above, $\sigma(i-1) = (i'_-, p_-, l_-)$ is often equal to $\sigma(i) - (0,0,1) = (i', p, l) - (0,0,1)$. In the same way, $\sigma(i+1) = (i'_+, p_+, l_+)$ is often equal to $\sigma(i) + (0,0,1) = (i', p, l) + (0,0,1)$. The affine relations that often occur are:

$$(i'_-, p_-, l_-) = (i', p, l) - (0,0,1) \wedge (i'_+, p_+, l_+) = (i', p, l) + (0,0,1)$$

We separate the domain D' into two sub-domains referred to as D_{steady} and $D_{boundary}$; the affine relations mentioned above (between $\sigma(i)$ and $\sigma(i-1)$, and between $\sigma(i)$ and $\sigma(i+1)$) hold everywhere in D_{steady}. For many codes of interest D_{steady} is much larger than $D_{boundary}$. Therefore, the code generated for D_{steady} avoids expensive integer computations. We generate code for the two domains:

$D_{steady} = \{(i, j_0, \ldots, j_n), i \in D, \sigma^{-1}(j_0) = u_0(i), \ldots, \sigma^{-1}(j_n) = u_n(i), j_0 = \alpha_0 j_\ell + \beta_0, \ldots, j_n = \alpha_n j_\ell + \beta_n\}$

$D_{boundary} = D' - D_{steady}$

with the same schedule as given above. The second domain allows us to scan the remaining, non-steady instances that corresponds to boundary conditions. As our generated code contains explicit parallel loops going over processor dimension p, we map each iteration of the p loop to a distinct physical processor to obtain parallel code.

V. Evaluation

Simulation Environment We simulated a 16-core tiled CMP with the Virtutech Simics full-system simulator [18] extended with timing infrastructure GEMS [19]. Each tile was a 4GHz 4-way in-order SPARC processor core, with 64KB split L1 instruction and data caches, a 512KB L2 bank and a router. An on-die 4×4 mesh connected the tiles, with 16GB one-way bandwidth per link. Cut-through routing was used in this packet-switched network. The latency on each link was modeled as 5 cycles. We had 4GB physical memory in our simulation configuration, which is larger than the memory requirement of any benchmark we used. DRAM access latency was modeled as 80ns and eight DRAM controllers were placed on the edge of the chip. The simulated hardware configuration is summarized in Table I.

The coherence protocol we simulated is very close to the *STATIC-BANK-DIR* protocol in [20]. We adopted a similar implementation from GEMS 1.4 and made a few changes for our experiments. With this protocol, a directory is distributed across all the tiles as shown in Fig. 1(b) such that each physical address has an implicit home tile. While our approach is not limited to this bank mapping scheme, the simulated coherence protocol interleaves cache lines.

Benchmark Suite While our approach can be applied in combination with program parallelization, we evaluated our approach with a set of data-parallel benchmarks as finding parallelism is not the focus of this paper. The selected benchmarks are described in Table II. Several of the benchmarks were taken from [21]. Iterative benchmarks, such as Skeleton, were set to execute 15 iterations.

Experimental Results We implemented the localization analysis framework by taking information (index set and array access functions) from LooPo's [23] dependence tester and generating statement-wise affine transformations. After localization analysis, the parallel code is generated by a tool based on CLooG, as presented in Sec. IV. We compiled generated C programs with GCC 4.0.4 using the "-O3" optimizing flag on a Sun SPARC Enterprise T5120 server. In order to verify the correctness of applied transformations, we wrote a validation procedure for every generated program, to compare with the result of a sequential implementation using canonical array layout. In our current implementation, we padded array elements to powers of two in bytes. For example, in Demosaic we used the RGBA format instead of the RGB format. An alternative solution to this problem is to have structures of arrays (SOA) instead of arrays of structures (AOS).

Table III summarizes the localization analysis results for the benchmarks. For each benchmarks examined, there exists at least a computation allocation and a data mapping that partially localize data accesses.

In order to assess the impact of computation allocation and data layout transformation, for each benchmark we generated two versions of parallel code:

1) We divided the iteration space of each statement into P chunks along the outermost parallel loop and used canonical data layout. This version corresponds to the code generated by the state-of-the-art production compilers for data-parallel languages. This version is referred to as *canon*.
2) Following the proposed localization analysis and code generation framework, we generated parallel code that uses non-canonical data layout with padding, strip-mining and permutation. This version is referred to as *noncanon*.

Benchmark	Communication free?	Fully localizable?	Partially localizable?	q
Sum	Yes			0
Mv	No	No	Yes	0
Demosaic	No	Yes		2
Convolve	No	No	Yes	2
Life	No	Yes		1
2D Jacobi	No	Yes		1
Skeleton	No	Yes		2

Table III
Localization analysis results of the benchmarks

Fig. 6 shows normalized remote L2 cache access numbers

Processor	16 4-way, 4GHz in-order SPARC cores
L1 cache	private, 64KB I/D cache, 4-way, 64-byte line, 2-cycle access latency
L2 cache	shared, 8MB unified cache, 8-way, 64-byte line, 8-cycle access latency
Memory	4GB, 320-cycle (80ns) latency, 8 controllers
On-chip network	4x4 mesh, 5-cycle per-link latency, 16GB/s bandwidth per link

Table I
SIMULATION CONFIGURATION

Benchmark	Description
Sum	Compute the sum of two 1000x1000 32-bit floating-point matrices.
Mv	Compute matrix-vector product between a 1000x1000 matrix and a vector.
Demosaic	Compute an RGB image from a 1000x1000 pixel Bayer pattern, more details in [21].
Convolve	Convolve a 1000x1000 monochromatic image with 5x5 Gaussian filter.
Life	Compute Conway's "Game of Life" on a 1000x1000 grid, more details in [21].
1D Jacobi	1D Jacobi computation with an iterative loop.
2D Jacobi	2D Jacobi computation with an iterative loop.
Skeleton	Compute the shape skeleton of a 2D object that is represented as non-zero pixels, used as an example in [22].

Table II
A DATA-PARALLEL BENCHMARK SUITE USED IN EVALUATION

compared to *canon* that uses canonical data layout. As illustrated in Fig. 6, code generated by the proposed framework significantly reduces the number of remote accesses. On average, with *noncanon* the number of remote L2 accesses is only 49% of that with *canon*. For iterative benchmarks including Life, 1D Jacobi, 2D Jacobi and Skeleton, this improvement is more significant, up to 0.039 compared with *canon*. This is because these benchmarks have most of their working sets staying in the L2 cache across iterations and benefit from greatly reduced L2 miss latencies.

Fig. 7 shows normalized on-chip network link utilization. On average, on-chip network traffic is reduced by 81.8% from *canon*. The most significant link utilization reduction is with Skeleton. The Skeleton code using non-canonical data layout only results in 2.39% of link utilization by the code using canonical layout. The above results have implications beyond performance. Power has become a first-order issue in chip design. Research has shown that the on-chip interconnect contributes to up to 50% of total power consumption of a chip [24]. Several hardware/software approaches have been proposed to reduce the energy consumption from the on-chip network. [25], [26], [27] Link utilization reduction illustrated in Fig. 7 essentially translates to reduction in dynamic power dissipation over the on-chip network.

Fig. 8 shows speedups relative to *canon*. We can observe significant performance improvements with our approach. The average speedup in our experiments is 1.64. The best speedup achieved is 2.83 with 1D Jacobi, which also has the most significant remote L2 access reduction.

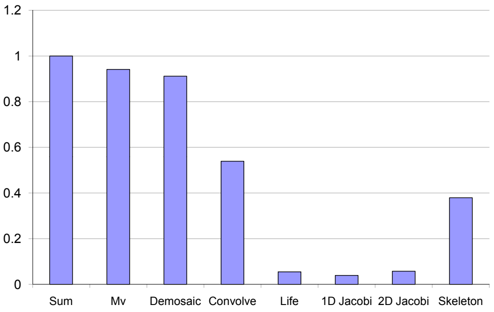


Figure 6. Normalized remote L2 cache access numbers with $\#remoteAccesses(canon) = 1$.

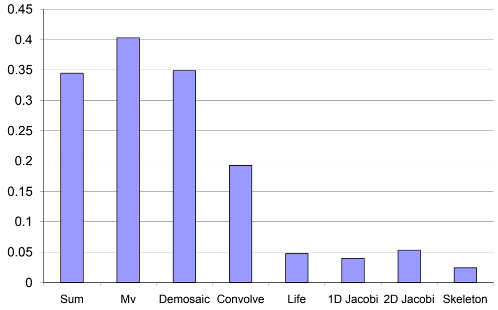


Figure 7. Normalized link utilization of on-chip network with $utilization(canon) = 1$

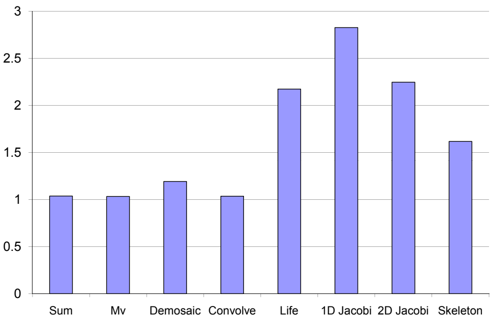


Figure 8. Speedup relative to code version *canon*.

VI. RELATED WORK

There has been a large body of prior work addressing the problem of avoiding remote memory accesses on NUMA/DSM systems, such as [28], [29]. Although our work is related in spirit, it significantly differs from prior work in two aspects. (i) NUCA architectures pose a very different performance issue than NUMA machines. In NUCA architectures, data with contiguous addresses are spread across L2 cache banks at a fine granularity by a static bank mapping function. Dynamic migration of cache lines between banks is not feasible due to its complexity and power consumption. In contrast, NUMA systems allocate memory at the page level by following the “first touch” rule or providing dynamic page allocation through software. This allows the programmer to control data distribution with canonical row-major or column-major layouts. (ii) To the best of our knowledge, none of the works provide a characterization as to when it is possible to achieve localizable computation allocation and data mapping.

There has been prior work attempting to use data layout optimizations to improve spatial locality in programs. Leung and Zahorjan [30] were the first to demonstrate cases where loop transformations fail and data transformations are useful. O’Boyle and Knijnenburg [31] presented techniques for generating efficient code for several layout optimizations such as linear transformations, memory layouts, alignment of arrays to page boundaries, and page replication. Kandemir et al. [32] presented a hyperplane representation of memory layouts of multi-dimensional arrays and show how to use this representation to derive very general data transformations for a single perfectly-nested loop. In the absence of dynamic data layouts, the layout of an array has an impact on the spatial locality characteristic of all the loop nests in the program which access the array. As a result, Kandemir et al. [33], [34], [32] and Leung and Zahorjan [30] presented a global approach to this problem; of these, [33] considered dynamic layouts. The motivating context and our approach to solution are fundamentally different from the above efforts.

Chatterjee et al. [35] presented a framework to determine array alignments in data-parallel languages such as HPF. While seemingly similar, the problem we address arises from the bank mapping imposed by the architecture and our approach to code transformation for non-canonical layouts is very different. Barua et al. [36] proposed to interleave consecutive array elements in a round-robin manner across the memory banks of the RAW processor. An optimization called modulo unrolling is applied to unroll loops in order to increase memory parallelism. Extending the approach by Barua et al. [36], So et al. [37] proposed to use custom data layouts to improve memory parallelism of FPGA-based platforms.

Rivera and Tseng [16] presented data padding techniques to avoid conflict misses. Within a different context, inter- and intra-array padding are employed by us to group elements into the same cache bank.

Chatterjee et al. [38], [39], [40] studied non-canonical data layouts such as 4D layout and different Morton layouts in matrix computations. The key idea is preserving locality from a 2D data space in the linear memory space. Although also employing non-canonical data layouts, our study differs from the above research in several aspects. The mismatch between data and memory space in our study is due to the banked cache organization so we may separate contiguous data in the memory space. Our focus on automatic code generation for general affine programs is very different from the above works.

Anderson et al. [15] employed non-canonical data layouts in compiling programs on shared-memory machines. Our work is related to theirs in that we also strip-mine array dimensions and then apply permutations. To avoid conflict misses and false sharing, the approach by Anderson et al. [15] maps data accessed by a processor to contiguous memory locations. In comparison, our approach attempts to spread data such that they are mapped to the same cache bank. Our data layout transformation approach shares some similarities with their approach, but our localization analysis and code generation framework using CLooG are very different.

Lim and Lam [41], [42] previously developed a formulation for near-neighbor communication; that formulation does not apply here nor can its adaptation be reused. This is due to its being a set-and-test approach that gives solutions only when neighbors within a fixed distance (independent of any parameters) can be found. In particular, the constraint is formulated as $\phi(i) - \phi(i') \leq \gamma$, where γ is a constant. The approach does not address the following: (i) what is the value to be used for γ?; (ii) how is γ varied when a solution is not found for a fixed γ; and (iii) how can the non-existence of any solution with such a constant γ (when the difference cannot be free of program parameters) be detected. Detecting the latter case is essential for feasibility of localization itself. Our approach solves all of these problems.

VII. Conclusions

Future chip multiprocessors will likely be based on a tiled architecture with a large shared L2 cache. The increasing wire delay makes data locality exploitation an important problem and it has been addressed by many hardware proposals. In this paper, we developed a compile-time framework for data layout transformation to localize L2 cache accesses using a polyhedral model for program transformation and code generation using CLooG. Significant improvements were demonstrated on a set of data-parallel benchmarks on a simulated 16-core chip-multiprocessor.

We believe that the approach developed in this paper provides the most general treatment to date of code generation for data layout transformation – analysis of data access affinities and automated generation of efficient code for the transformed layout. Programs with multiple statements in imperfectly nested loops are handled, as well as multiple arrays with different affine indexing functions. An application of the approach to layout transformation and code generation is currently targeting enhanced vectorization of stencil computations.

Acknowledgment

This research is supported in part by the U.S. National Science Foundation through awards 0403342, 0811781 and 0509467. Sriram Krishnamoorthy contributed to this work when he was a PhD student at The Ohio State University. Christophe Alias contributed to this project when he was a postdoctoral researcher at The Ohio State University. Haibo Lin contributed to this work when he worked at Intel China

Research Center. The authors would also like to thank the anonymous reviewers for their comments on the earlier version of this paper.

References

[1] R. N. Kalla, B. Sinharoy, and J. M. Tendler, "IBM POWER5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, 2004.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *ASPLOS'02*.

[4] B. M. Beckmann and D. A. Wood, "Managing wire delay in large chip-multiprocessor caches," in *MICRO'04*.

[5] B. S. Andersen, F. G. Gustavson, A. Karaivanov, M. Marinova, J. Waniewski, and P. Y. Yalamov, "Lawra: Linear algebra with recursive algorithms," in *PARA '00*. London, UK: Springer-Verlag, 2001, pp. 38–51.

[6] P. Feautrier, "Some efficient solutions to the affine scheduling problem: I. one-dimensional time," *IJPP*, vol. 21, no. 5, pp. 313–348, 1992.

[7] W. Pugh and D. Wonnacott, "Constraint-based array dependence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 635–678, 1998.

[8] W. Kelly and W. Plugh, "Minimizing communication while preserving parallelism," in *ICS '96*. New York, NY, USA: ACM, 1996, pp. 52–60.

[9] F. Quilleré, S. V. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *Intl. J. of Parallel Programming*, vol. 28, no. 5, pp. 469–498, 2000.

[10] "CLooG: The Chunky Loop Generator," http://www.cloog.org.

[11] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *PLDI'08*, Tucson, Arizona, June 2008.

[12] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI '08*. New York, NY, USA: ACM, 2008, pp. 101–113.

[13] "GCC: GCC, the GNU Compiler Collection, Version 4.3.2," http://gcc.gnu.org.

[14] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004, habilitation Thesis. [Online]. Available: http://www.uni-passau.de/~griebl/habilitation.html

[15] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and computation transformations for multiprocessors," in *PPOPP'95*.

[16] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," in *PLDI'98*.

[17] S. Amarasinghe, *Paralelizing Compiler Techniques Based on Linear Inequalities*. Stanford University, 1997, phD Dissertation. [Online]. Available: http://suif.stanford.edu/papers/amarasinghe97.ps

[18] Virtutech AB, "Simics full system simulator," http://www.simics.com.

[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[20] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *ISCA'07*.

[21] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *ASPLOS'06*.

[22] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A machine independent programming language for parallel computers," *IEEE TSE*, vol. 26, no. 3, pp. 197–211, 2000.

[23] "LooPo - Loop parallelization in the polytope model," http://www.fmi.uni-passau.de/loopo.

[24] N. Magen, A. Kolodny, U. Weiser, and N. Shamir, "Interconnect-power dissipation in a microprocessor," in *SLIP'04*, 2004.

[25] L. Shang, L.-S. Peh, and N. K. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," in *HPCA'03*.

[26] G. Chen, F. Li, M. Kandemir, and M. J. Irwin, "Reducing NoC energy consumption through compiler-directed channel voltage scaling," in *PLDI'06*.

[27] F. Li, G. Chen, M. Kandemir, and I. Kolcu, "Profile-driven energy reduction in network-on-chips," in *PLDI'07*.

[28] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "A case for user-level dynamic page migration," in *ICS'00*.

[29] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in *ASPLOS'94*.

[30] S. Leung and J. Zahorjan, "Optimizing data locality by array restructuring," Dept. Computer Science, University of Washington, Seattle, WA, Tech. Rep. TR-95-09-01, 1995.

[31] M. F. P. O'Boyle and P. M. W. Knijnenburg, "Non-singular data transformations: definition, validity, applications," in *CPC'96*.

[32] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A linear algebra framework for automatic determination of optimal data layouts," *IEEE TPDS*, vol. 10, no. 2, pp. 115–135, 1999.

[33] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade, "Static and dynamic locality optimizations using integer linear programming," *IEEE TPDS*, vol. 12, no. 9, pp. 922–941, 2001.

[34] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *MICRO'98*.

[35] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng, "Automatic array alignment in data-parallel programs," in *POPL'93*. New York, NY, USA: ACM Press, 1993, pp. 16–28.

[36] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: a compiler-managed memory system for raw machines," in *ISCA'99*.

[37] B. So, M. W. Hall, and H. E. Ziegler, "Custom data layout for memory parallelism," in *CGO'04*.

[38] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive array layouts and fast parallel matrix multiplication," in *SPAA'99*.

[39] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *ICS'99*.

[40] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," in *HPCA'00*.

[41] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine partitions," *Parallel Computing*, vol. 24, no. 3-4, pp. 445–475, 1998.

[42] A. W. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *ICS'99*, 1999, pp. 228–237.