A Compile-Time Data Locality Optimization Framework for NUCA Chip Multiprocessors

Qingda Lu, Uday Bondhugula, Sriram Krishnamoorthy, P. Sadayappan
Dept. of Computer Science and Engineering
The Ohio State University

J. Ramanujam
Dept. of Electrical and Computer Engineering
Louisiana State University

Yongjian Chen, Haibo Lin, Tin-Fook Ngai
Intel Corporation

Technical Report
OSU-CISRC-6/08-TR29

A Compile-Time Data Locality Optimization Framework for NUCA Chip Multiprocessors

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. For data-parallel programming models, there is a mismatch between such a non-uniform cache organization and the canonical row-major or column-major layouts of multi-dimensional arrays – causing a significant number of non-local L2 accesses for many commonly occurring data access patterns. In this paper we develop a compile-time framework for data locality optimization via data layout transformation. Using a polyhedral model for dependences, the program's localizability is determined by analysis of intra- and inter-statement dependences, 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.

1. Introduction

Most proposed chip multiprocessor (CMP) designs [39, 17, 22] 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 [21, 15, 3]. Fig. 1 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. Such an architecture is highly scalable and is embraced by Intel's Tera-project [16] that has demonstrated the capability of integrating 80 processors on a single die. Future CMPs are likely to be based on a similar design to enable large numbers of processors and large on-chip caches through emerging technologies such as 3D stacking. Fig. 2 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. So 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. The evaluations in this paper assume cache-line interleaving, but the developed approach is also applicable to coarser interleaving across banks.

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

Let us consider the code in Fig. 3, 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. We consider the following mapping schemes with one thread on each processor: (i) based on the *owner-computes* rule,

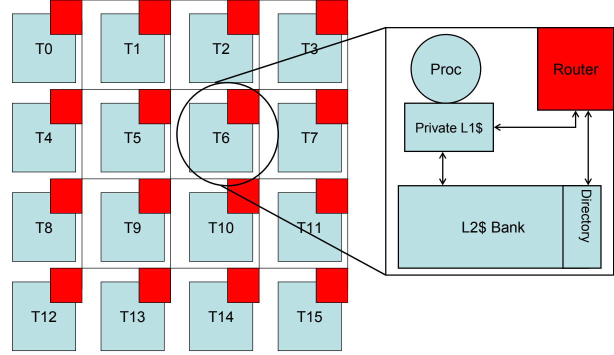
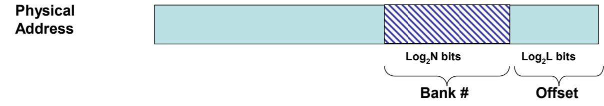


Figure 1. Tiled CMP architecture



N: number of L2 cache banks
L: bank offset size, some power of 2 between cache line and page size

Figure 2. Mapping a physical address to a cache bank

```
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 }
```

Figure 3. 1D Jacobi code.

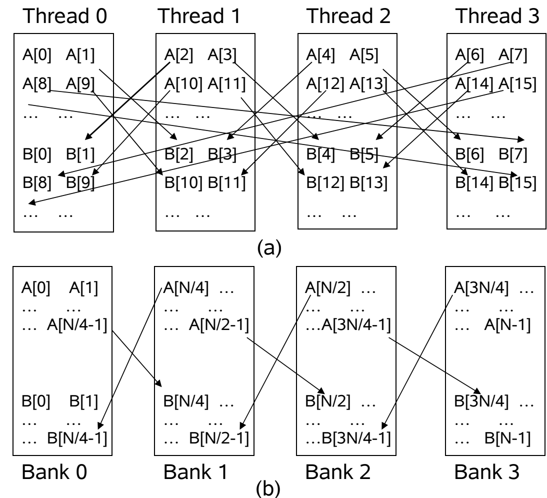


Figure 4. Parallelizing a 1D Jacobi program on a 4-tile CMP.

each thread is responsible for computing the data mapped to its local bank – this corresponds to OpenMP's static scheduling with a chunk size of L; (and) (ii) 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. We observe that aligning the base addresses of A and B is important since A[i] and B[i] are always referenced together. Let us assume that the origins of A and B are both aligned to bank 0's boundary. Fig. 4(a) illustrates 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, to keep the affinity between computation and data. This is shown in Fig. 4(b). As illustrated, there are only six remote accesses in every outer iteration.

The above example illustrates the importance of an integrated computation/data mapping scheme for achieving low access latencies on future tiled CMPs. With canonical row-major or column-major layouts, the block-cyclic mapping imposed by bank address interpretation often results in significant on-chip communication due to remote bank L2 cache access. To overcome the mismatch between a flat memory model and the banked L2 cache organization on a tiled CMP, non-canonical data layouts such as the one in Fig. 4(b) may need to be utilized. In this paper, we develop a compile-time framework for determining such data layout transformations. We show how the feasibility of bank localization can be formulated in a polyhedral model for program transformations [23] and how code can be generated using optimized non-canonical layouts.

Although several papers from the architecture community have investigated NUCA issues with CMPs [21, 15, 3, 11], to the best of our knowledge, this paper represents the first attempt at a systematic formalization of compiler analysis for data locality optimization in the context of NUCA CMP's. The rest of the paper is organized as follows. We first provide some motivation and background for our work. We then present the analysis framework that determines the feasibility of data localization and computation / data mappings in Sec. 3. In Sec. 4, we describe the code generation framework for creation of transformed programs with non-canonical layout. We evaluate the effectiveness of our approach in Sec. 5 with an architecture simulator using several benchmarks. We discuss related work in Sec. 6 and present our conclusions in Sec. 7.

2. Background and Overview of Approach

Most commercial designs [39, 17, 22] and research projects have employed a shared L2 cache on a CMP to emphasize minimizing 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 and connects these banks through an on-chip network [21, 15, 3]. Prior NUCA work has proposed approaches to distributing data across L2 cache banks. Static NUCA (S-NUCA) schemes derive the bank index from several contiguous bits of a physical address as demonstrated in Fig. 2. With different address interpretations, S-NUCA's interleaving granularity varies from cache line to page frame. 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. The Sun UltraSPARC T1 [22] uses a cache-line interleaved mapping scheme in its banked L2 cache.

A data-parallel programming model provides aggregate data types such as arrays and operations on them. Examples of existing data-parallel languages include Fortran 90/95, HPF, and ZPL. Because of its high abstraction level, a data-parallel programming model hides implementation details and provides ease of programming and portability. Parallelism is specified implicitly with a data-parallel model and exploited by having tasks operating on an object collectively. Tarditi et al. [38] summarize commonly used operations of type *Array*. Note that these operations are not limited to element-wise operations as reduction and data transformation operations are also included.

While the data-parallel programming model has been studied for a long time, with the advent of CMPs and general-purpose GPU computing, great interest has been rekindled because of the expressiveness and effectiveness of the data-parallel programming model. Proposed HPCS (High-Productivity Computing System) languages all provide data-parallel operations. Several recent research and commercial projects [38, 33, 32] have also proposed data-parallel languages or libraries and studied how to map data-parallel programs to CMPs or GPUs. In this paper we address the issue of efficiently mapping data-parallel programs to tiled CMPs.

2.1 Overview of Our Approach

Although CMPs provide a shared memory model, exploitation of data locality is crucial for NUCA-CMPs. In order to reduce inter-processor data movement costs in accessing the shared cache, rearrangement of array data layout may be essential to ensure that most L2 accesses are on local or neighboring cache banks.

The key data type in data-parallel programs is the multi-dimensional array. With such a programming model, parallelism is specified implicitly and functional semantics is used. Memory management and array accesses are transparent to the programmer by having a managed vector memory space. Therefore the compiler has complete freedom in deciding how to map the computation to processor cores and in choosing the data layout for each array.

The example in Fig. 3 illustrates the benefit of data layout transformation. The key idea behind the transformation is data localization onto cache banks, to match iteration distribution among processors. We seek to answer the following questions:

When is such a transformation beneficial? How can the transformation be automated?

Using the polyhedral framework for program transformation, we develop an approach to characterizing the feasibility of data localization on cache line interleaved NUCA systems, and determination of the needed transformation. The developed compile-time optimization framework consists of two phases: localization analysis and code generation. In the localization analysis phase we use a polyhedral loop transformation framework to determine whether a program can be localized. The localization analysis phase performs a global analysis of the program, to identify: (1) arrays that can be localized for communication-free accesses (e.g. summation between vectors), and (2) arrays that can be localized via layout transformation for communication-optimized access (similar to those in Fig. 3). This is done by formulating the problem as that of finding suitable affine partitioning hyperplanes in a polyhedral representation of the iteration space(s) of the program. The localization phase is followed by the code generation phase that employs the data and computation mappings to engender localization and transform the code to support non-canonical data accesses.

3. Localization Analysis

In this section, we describe the procedure to determine the localizability of a given program. Our analysis is based on the polyhedral framework since most data-parallel programs are regular and satisfy its affine conditions. The code generation aspects are discussed in Sec. 4.

3.1 Polyhedral Framework for Program Transformation

We begin with background information on the polyhedral model for loop transformation, which is the basis of our localization analysis framework. The notation used is similar to that used by Griebl [14].

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 $\mathbf{A}_{S,A}(\vec{i})$. A dynamic (or run-time) instance of a statement S at the iteration vector $\vec{i}$ is represented as $< S, \vec{i} >$. 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 [13]. 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* [13] 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. After such transformations π_S and ψ_A can be determined by using a linear programming solver.

LEMMA 1 (Farkas Lemma). *Let $\mathcal{D}$ be a non-empty polyhedron defined by p affine inequalities*

$$a_j x + b_j \geq 0, \;\; 1 \leq j \leq p.$$

Then, an affine form f is non-negative everywhere in $\mathcal{D}$ if and only if it is a positive affine combination:

$$f \equiv \lambda_0 + \sum_{j=1}^{p} \lambda_j (a_j x + b_j), \; \lambda_j \geq 0 \wedge 0 \leq j \leq p. \quad (1)$$

The non-negative constants λ_j are referred to as Farkas multipliers.

3.2 Localization Constraints

We define a program as *localizable* if there exists a computation allocation and a data mapping to a processor grid such that the data required by any processor is available in the neighborhood of that processor in the grid. Such a mapping reduces global communication, potentially reducing the total non-local data accesses. The dimensionality of the processor grid can be independent of the physical organization of a tiled CMP without impacting the benefits of localization. In this paper, we consider one-dimensional virtual processor grids. The localizability constraints for computation allocation are formalized as follows.

DEFINITION 1 (Localized computation allocation). *For a given program P, let h-transformation $h_{SS'}$ represent a data dependence between statements S and S' on array references $\mathbf{A}_{S,A}$ and $\mathbf{A}_{S',A}$ and P_e denote the domain of $h_{SS'}$. An affine computation allocation π for P is localized if and only if for any statements S and S' with data dependences e,*

$$\forall \vec{i'},\; P_e(\vec{i'}) \geq 0 \wedge \vec{i} = h_{SS'}(\vec{i'}) \implies |\pi_S(\vec{i}) - \pi_{S'}(\vec{i'})| \leq k, \quad (2)$$

where k is a constant.

With a localized computation allocation, Def. 2 provides constraints of localized data mappings.

DEFINITION 2 (Localized data mapping). *For a program P, let D be its index set and π be a localized computation allocation. An affine data mapping ψ for P is localized if and only*

if for any array A, and any reference $\mathbf{A}_{S,A}$*,*

$$\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies |\pi_S(\vec{i}) - \psi_A(\mathbf{A}_{S,A}(\vec{i}))| \leq q, \tag{3}$$

where q is a constant.

As a special case case, *communication-free localization* can be achieved if the constraints in Lemmas 2 and 3 can be satisfied.

LEMMA 2 (Communication-free computation allocation). *For a given program P, let h-transformation* $h_{SS'}$ *represent a data dependence between statements S and S' on array references* $\mathbf{A}_{S,A}$ *and* $\mathbf{A}_{S',A}$ *and* P_e *denote the domain of* $h_{SS'}$*. An affine computation allocation* π *for P is communication-free if and only if for any statements S and* S' *with data dependences,*

$$\forall \vec{i'},\ P_e(\vec{i'}) \geq 0 \wedge \vec{i} = h_{SS'}(\vec{i'}) \implies \pi_S(\vec{i}) - \pi_{S'}(\vec{i'}) = 0, \tag{4}$$

LEMMA 3 (Communication-free data mapping). *A program with communication-free computation allocation* π *is communication-free if and only if for any array A and any array reference* $\mathbf{A}_{S,A}$ *in a statement S, data mapping* ψ *satisfies*

$$\psi_A(\mathbf{A}_{S,A}(\vec{i})) = \pi_S(\vec{i}) \tag{5}$$

3.3 Localization Analysis Algorithm

Our localization analysis algorithm based on the above discussion consists of the following steps.

Step 1: Grouping Interrelated Statements/Arrays We determine connected sets of data-parallel statements in an input program. Given a data-parallel program, we form a bipartite graph where each vertex corresponds to a data-parallel 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: Rewriting Array Indices We rewrite the program such that all array references are to byte arrays. For example, a reference $A[i][j+1]$ to a double array A is rewritten as $A_{byte}[i][8j+8]$. This allows us to account for different data sizes since the default layout maps data based on their byte addresses. This is explained in detail in Sec. 4.

Step 3: Finding Localized Computation Mapping Following Def. 1, we formulate the problem as finding an affine computation allocation π that satisfies $|\pi(\vec{i}) - \pi(\vec{i'})| \leq k$ for every pair of iterations i and i' that access a common data element. The π function identifies a parallel space dimension in the iteration space of each data parallel statement. Therefore a system of such equalities are collected as constraints for all data dependences including input dependences.

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

$$\forall \vec{i'},\ P_e(\vec{i'}) \geq 0 \implies f(\vec{i'}) + k \geq 0 \tag{6}$$

$$\forall \vec{i'},\ P_e(\vec{i'}) \geq 0 \implies -f(\vec{i'}) + k \geq 0 \tag{7}$$

where $\forall \vec{i'},\ f(\vec{i'}) = \pi_S(h_{SS'}(\vec{i'})) - \pi_{S'}(\vec{i'})$. For simplicity of presentation, we refer to $f(\vec{i'}) + k$ and $-f(\vec{i'}) + k$ as $f_1(i')$ and $f_2(i')$ respectively.

We apply Farkas Lemma to transform the above constraints to affine equalities by introducing Farkas multipliers. Take Eq. 6 as an example. With Farkas Lemma, we have $\forall \vec{i'},\ P_e(\vec{i'}) \geq 0 \implies f_1(\vec{i'}) \geq 0$ if and only if $f_1(\vec{i'}) \equiv \lambda_0 + \sum_k \lambda_k(a_k\vec{i'} + b_k)$ with $\lambda \geq 0$, where affine inequalities $a_k\vec{i'} + b_k \geq 0$ define P_e. Therefore we have the following:

$$M'_{f_1} \begin{bmatrix} \vec{i'} \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda_1 & \ldots & \lambda_m, & \lambda_0 \end{bmatrix} \begin{bmatrix} M'_{P_e} \\ 0 \ldots 01 \end{bmatrix} \begin{bmatrix} \vec{i'} \\ 1 \end{bmatrix} \tag{8}$$

where $f_1(\vec{i'}) = M'_{f_1} \begin{bmatrix} \vec{i'} \\ 1 \end{bmatrix}$ and $P_e(\vec{i'}) = M'_{P_e} \begin{bmatrix} \vec{i'} \\ 1 \end{bmatrix}$. Since Eq.(8) holds for all $\vec{i'}$, we have

$$M'_{f_1} = \begin{bmatrix} \lambda_1 & \ldots & \lambda_m, & \lambda_0 \end{bmatrix} \begin{bmatrix} M'_{P_e} \\ 0 \ldots 0\ 1 \end{bmatrix} \tag{9}$$

We apply Fourier-Motzkin elimination to Eq. (9) to remove all the Farkas multipliers. The new set of constraints are in the form of $M'_{f_1} G_1 \geq 0$. The whole procedure is also applied to Eq. (7) and we obtain $M'_{f_2} G_2 \geq 0$. These two together form the set of inequalities that constrain the coefficients of π.

In order to minimize potential communication, we solve the system of inequalities generated as an integer programming problem. $min(k)$ is used as the objective function to minimize the communication overhead. If such a k is identified, we find a localizable computation allocation.

As a special case, if k is determined to be 0 the computation is identified to have communication-free computation localization. If a non-zero k is determined as the minimal solution, the program is not communication-free but is localizable.

If a communication-free π is found, we decide each array's data mapping based on Eq. (5). For an array A, for each of its reference $\mathbf{A}_{S,A}$, we collect linear equalities:

$$M'_{\psi_A} \begin{bmatrix} M'_{\mathbf{A}_{S,A}} \\ 0 \ldots 0\ 1 \end{bmatrix} = M'_{\pi_S} \tag{10}$$

We can decide data mapping for A by solving this system of equalities collected from every reference of A.

If a localized but not communication-free computation allocation π is found, we find a localized data mapping ψ based on Def. 2. For each array reference in statement S, we collect constraints in Eq. (3) under S's index set. Similar to finding a localized computation allocation, we transform these constraints to linear inequalities using Farkas Lemma and Fourier-Motzkin elimination. With objective function $min(q)$, we have a linear programming problem to solve.

If a solution is not obtained by the above steps, the program is identified as non-localizable. Traditional iteration-reordering transformations are then performed to optimize for temporal locality without changing the data layout.

Summary The algorithm is summarized in Algorithm 1.

Partial Localization We also consider partial localization. That is, we allow a processor to have a small number of data accesses outside its neighborhood. If a localized π cannot be found, we attempt to find partial localization by removing

Algorithm 1 Localization analysis algorithm

Require: Generalized dependence graph (including h-transformations, dependence polyhedra, and access functions) after indices are rewritten to access byte arrays

1: $C = \phi$
2: **for** each dependence e (including input dependence) **do**
3: Obtain new constraints: $\pi(\vec{i}) - \pi(h(\vec{i})) + k \geq 0$ and $\pi(h(\vec{i})) - \pi(\vec{i}) + k \geq 0$ under $\vec{i} \in P_e$.
4: Apply Farkas Lemma to new constraints to obtain linear constraints and eliminate all Farkas multipliers
5: Add linear inequalities from the previous step into C
6: **end for**
7: Add objective function $(\min k)$
8: Solve the result linear programming problem with constraints in C
9: **if** solution π is found **then**
10: **if** k=0 **then**
11: **for** each array A **do**
12: $C = \phi$
13: **for** each reference $\mathbf{A}_{S,A}$ **do**
14: Add linear constraints from Eq. (10) to C (based on Lemma 3)
15: **end for**
16: Solve the system of equalities in C to find ψ_A
17: **end for**
18: **if** ψ is found for each array **then**
19: return π and ψ
20: **end if**
21: **end if**
22: $C = \phi$
23: **for** each array refrence $\mathbf{A}_{S,A}$ **do**
24: Obtain new constraints: $\pi_S(\vec{i}) - \psi_A(\mathbf{A}_{S,A}(\vec{i})) + q \geq 0$ and $\psi_A(\mathbf{A}_{S,A}(\vec{i})) - \pi_S(\vec{i}) + q \geq 0$ under $\vec{i} \in D_S$.
25: Apply Farkas Lemma to new constraints to obtain linear constraints and eliminate all Farkas multipliers
26: Add linear inequalities from the previous step into C
27: **end for**
28: Add objective function $(\min q)$
29: Solve the result linear programming problem with constraints in C
30: **if** ψ is found **then**
31: return π and ψ
32: **end if**
33: **end if**
34: return "not localizable"

constraints from input dependences on small arrays with high reuse. Then we recompute π, seeking localization with the reduced constraints – localization of small arrays being ignored in the expectation that temporal locality may be exploited by the private L1 cache.

As an example of partial Localization, code to compute matrix-vector multiplication is shown in Fig. 5. There are two sets of data dependences between instances (i, j) and (i', j') of statement 2:

- output dependence on array C: $i - i' = 0, j - j' \geq 1$
- input dependence on array B: $i - i' \geq 1, j - j' = 0$

We are not able to find a localized computation allocation considering both dependences. To overcome this problem we remove the input dependences on array B because B's reuse

```
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 5. Matrix-vector multiplication code.

is asymptotically higher than A's and its size is asymptotically smaller. We are then able to obtain $\pi(i, j) = i$ that is communication-free regarding arrays A and B.

Using the Linear Programming Solver The linear programming solver (Parametric Integer Programming [12]) that we use works in the integer space and uses lexicographic ordering instead of an objective function as its ranking principle. It has been used by us to implement Step 3 of the localization algorithm with the following consideration.

- In order to reduce the space overhead introduced by padding (discussed in Sec. 4), we attempt to find a parallel loop accessing slow-varying array dimensions. We order π's unknown coefficients increasingly based on their ranks, which is the slowest-varying dimension a given loop index appears at. For example, if a loop index i appears in $A[i]$, $B[i][j]$ and $C[i][j][k]$, its unknown coefficient is ranked 3 from $C[i][j][k]$. Furthermore, for unknowns with the same rank, they are ordered based on their occurrences.
- To find a localized computation allocation, instead of supplying $\min(k)$ as an objective function, we have k as the first unknown. The same approach is used to find a localized data mapping.
- While using a solver in the integer space, we do expect to obtain rational coefficients such as 1/2 for a localized data mapping ψ. In order to do so with a localized computation allocation π, we rewrite Eq. 3 in Def. 2 as

$$\forall \vec{i},\ D_S(\vec{i}) \geq 0 \implies |\mu \cdot \pi_S(\vec{i}) - \psi_A(\mathbf{A}_{S,A}(\vec{i}))| \leq q,\ \textit{where}\ \mu \geq 1 \quad (11)$$

Now we introduce a new positive unknown μ which essentially stretches the image of function π. We can find μ and a localized ψ' using the linear programming solver. The data mapping with the original system is $\psi = \frac{1}{\mu}\psi'$.

4. Code Generation

In this section, we introduce the framework that generates code after the localization analysis framework determines computation allocation and data mapping.

4.1 Virtual-Physical Processor Mapping

Based on the localization analysis results from the previous section, here we discuss the mapping τ between virtual processors and P physical processor cores on a NUCA-CMP.

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 $\psi_A(\vec{d}) = d_k/s_A + off_A$, 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. We determine the mapping between virtual processors and physical processor cores based on data mappings as follows.

If we find communication-free localization, where all arrays share the same stride along their fastest varying dimensions, we map virtual processors to physical processors in a block-cyclic fashion. Because all array references have been rewritten to access byte arrays, the block size is L/s, where L is the cache line size in bytes and s is the stride; we have $\tau(x) = \lfloor \frac{x}{L/s} \rfloor \bmod P$. In this fashion, we match data and computation on each processor while keeping the canonical data layout. Otherwise virtual processors are grouped into P blocks if the computation is localizable. Assuming the image of function f is the range of $[min_f, max_f]$, we have $\tau(x) = \lfloor \frac{x-\alpha}{(\beta-\alpha)/P} \rfloor$, where $\alpha = \min(min_\pi, min_\psi)$ and $\beta = \max(max_\pi, max_\psi)$.

For the 1D Jacobi example in Fig. 3, the localization analysis determines the program to be localizable but not communication-free and obtains $\pi(i) = i$ and $\psi(i) = i$. From the above procedure, we have the virtual-physical processor mapping as $\tau(x) = \lfloor \frac{x}{N/P} \rfloor$.

4.2 Data Mapping Enforcement

For any array A in a localizable data-parallel program, we can obtain data mapping $\tau(\psi(\vec{d}))$ that maps array elements to physical processors. However, on a tiled CMP with a shared memory, data mappings cannot be through explicit data distribution. Here we apply data layout transformations including *strip-mining*, *permutation* and *padding* to enforce data mappings on tiled CMPs. These techniques have been used by previous studies [1, 34, 35] to avoid conflict misses or reduce false sharing. In comparison here we employ them to improve distance locality on a tiled CMP.

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 $(..., N_1, ..., N_2, ...)$ is transformed to $(..., N_2, ..., N_1, ...)$ and an array reference $[...][i_1][...][i_2][...]$ becomes $[...][i_2][...][i_1][...]$. Combined with strip-mining, permutation changes an array's actual layout in the memory space. Fig. 6 shows how one-dimensional array A in the 1D Jacobi example in Fig. 3 is transformed with strip-mining and permutation to realize its block-distributed data mapping. Fig. 6(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 the block-distributed data mapping, 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 $A[i]$ as $A'[i_p][i_c][i_L]$. As shown in Fig. 6(b), by permuting the first two dimensions of A', we obtain a new array A'' and we are able to map $A'[i_p][i_c][i_L]$(i.e., $A''[i_c][i_p][i_L]$) to tile i_p such that block distribution of A is achieved.

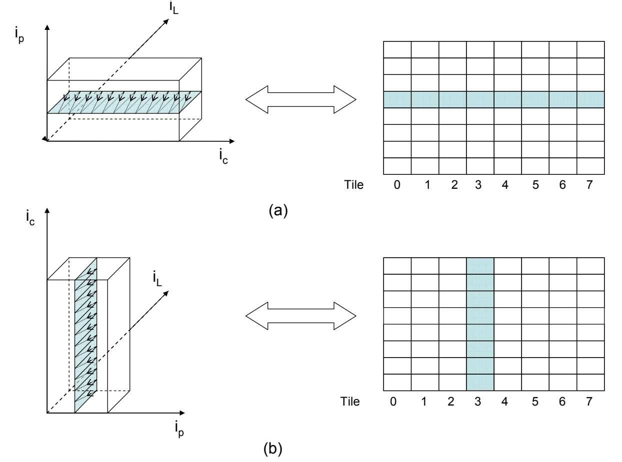


Figure 6. A one-dimensional example of data transformations. (a) Canonical layout after strip-mining. (b) Non-canonical layout after strip-mining and permutation.

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*. 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. 6, we pad N to $N' = \lceil \frac{N}{PL} \rceil \cdot P \cdot L$ and the last $N' - N$ elements are ignored in the computation.

All data layout transformations discussed so far are not limited to one-dimensional arrays or the fastest varying dimension. For example, if we follow localization analysis results to create a blocked view of array $A(N_1, N_2)$ along its first dimension, both dimensions are strip-mined to get $A'(P, \lceil \frac{N_1}{P} \rceil, \lceil \frac{N_2}{L} \rceil, L)$. Padding is needed if $P \nmid N_1$ or $L \nmid N_2$. Finally after a permutation involving the first three dimensions, we obtain $A''(\lceil \frac{N_1}{P} \rceil, \lceil \frac{N_2}{L} \rceil, P, L)$ that maps contiguous rows to a tile.

At the beginning and the end of the generated code, explicit data transformations may be needed to enforce data mappings of input and output arrays. Such transformation code can be found in Fig. 7, which outlines the generated code for the 1D Jacobi example. Such transformation code can often be fused with the first use of an input array and the last write of an output array. Also as shown in the code, we choose to linearize array indices instead of using multidimensional views. It is because we want generated code to work with different architectural

```
// threaded SPMD code
compute(int tid) {
  processor_bind(tid);
  B = ceiling(N/(P*L))*L;
  // transform input array A
  for (i = tid*B; i < min(N, (tid+1)*B); i++) {
    A'[index(i)] = A[i]; }
  while (condition) {
    for (i = tid*B; i < min(N, (tid+1)*B); i++) {
      B'[index(i)] = A'[index(i-1)] + A'[index(i)]
                   + A'[index(i+1)]; }
    synchronize(tid);
    for (i = tid*B; i < min(N, (tid+1)*B); i++) {
      A'[index(i)] = B'[index(i)]; }}
  // transform output array B
  for (i = tid*B; i < min(N, (tid+1)*B); i++) {
    B[i] = B'[index(i)]; }}
// index calculation for A, B and C
int index(int i) {
  N' = ceiling(N/(P*L))*P*L;
  //linearize ((i mod (N'/P))/L, i/(N'/P), i mod L);
  index = (i mod (N'/P))/L*P*L + i/(N'/P)*L + i mod L;
  return index; }
```

Figure 7. Outline of the generated code for 1D Jacobi.

configurations while most C compilers do not provide variable-length array support as in the C99 standard.

4.3 Computation Allocation Enforcement

We generate SPMD-style parallel code with P threads to enforce computation allocation decisions. Each thread is bound to its assigned processor through a system call and is responsible for computing the iterations mapped to this processor by physical computation allocation $\tau(\pi(\vec{i}))$. Fig. 7 outlines such generated code to compute the 1D Jacobi example.

4.4 Other Optimizations

Index Calculation The most severe problem with the code in Fig. 7 is its high indexing cost. It is not feasible to compute the index of every array reference with modulo and division. With regular access functions in the original program, we apply a set of optimizations similar to those in [1] to reduce the indexing overhead, which can be viewed as one form of strength reduction. One key observation is that within a certain range, a transformed indexing function is still an affine expression. For example in Fig. 7,for a given k, elements within the range $kL \le i \le kL+L-1$ can be accessed sequentially. Once crossing the boundary of strip-mined dimensions, we can also avoid the complex index computation by finding a large constant stride. One example with the code in Fig. 7 is that the index is incremented by $PL-(L-1)$ when i is increased from $kL-1$ to kL. Another optimization is if a statement has array references across boundaries of strip-mined dimensions in some iterations, it is possible to separate these iterations with the rest using loop peeling. Beside above optimizations used in [1], two additional optimizations are applied: (1) Because P and L are in most cases powers of two, bitwise operations such as SHIFT and AND are used to replace expensive multiplication, modulo and division operations whenever possible. (2) If array elements are accessed sequentially, loops are strip-mined in accordance to data strip-mining to reduce overhead from boundary checks. The code in Fig. 8 shows how the last loop in Fig. 7 is transformed to reduce the indexing cost.

L1 Cache Buffering If communication-free localization is achieved, every processor only accesses data mapped to its

```
ii_bound  = min(N, (tid+1)*B);
index = (tid*B) / (N'>>logP)) << logL;
for (ii = tid*B; ii < ii_bound; ii=ii+L) {
  for (i = ii; i < min(ii+L, ii_bound); i++,index++) {
    C[i] = C'[index]; }
  index = index + P*L - L; }
```

Figure 8. Optimizing index calculation.

local L2 cache bank. While distance locality is exploited for the L2 cache, two problems arise with L1 cache accesses. First, a private L1 cache is co-divided with the shared L2 cache into P partitions, therefore only $1/P$ of the L1 cache capacity can be exploited for data reuse. Second, because data mapped to the same L2 cache bank share low-order index bits, the possibility of conflict misses significantly increases. While this problem is with both L1 and L2 accesses, it is more severe with L1 cache due to its lower degree of associativity. Note that this problem also arises with general localization and does not depend on whether a cache is virtually- or physically-addressed because bank interleaving is within a page frame.

To overcome the problems with L1 accesses, we copy portions of arrays into pre-allocated buffers (referred to as *L1 buffers*) before they are actually used, if there is high data reuse. Because L1 buffers are contiguous in memory, the L1 cache's full capacity is exploited as long as the L1 buffers fit in L1 cache. Conflict misses are also avoided with these address-contiguous buffers with padding. Moreover, the utilization of L1 buffers benefits index calculation. When accesses are across boundaries of strip-mined array dimensions, division and modulo are often not avoidable in computing the indices. However, with L1 cache buffering, the expensive operations are eliminated in the copying step and indexing L1 buffers introduces little overhead due to their canonical layouts.

It is challenging to bound array sections accessed by arbitrary references within a data tile. We therefore limit L1 cache buffering to read-only arrays with group reuse. In practice, most of the index functions are single index variables (SIV), which further simplifies the task. For example, in a 2D Jacobi program, the source array $A(N,N)$ is accessed with references $A(i,j-1)$, $A(i,j+1)$, $A(i-1,j)$, $A(i+1,j)$. Assuming we block-distribute A along dimension i and tile loop i with a tile size TS, we copy array section from row $i-1$ to row $i+TS$ into L1 buffer $bufA(TS+2,N)$ before the intra-tile loop consumes buffered data.

5. Evaluation

5.1 Simulation Environment

We simulated a 16-core tiled CMP with the Virtutech Simics full-system simulator [40] extended with timing infrastructure GEMS [28]. 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 con-

trollers were placed on the edge of the chip. The simulated hardware configuration is summarized in Table 5.1.

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 bandwidth per link

Table 1. Simulation Configuration

The coherence protocol we simulated is very close to the *STATIC-BANK-DIR* protocol in [29]. 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. 2 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.

5.2 Benchmark Suite

We evaluated our approach with a set of data-parallel benchmarks, described in Table 5.2. Several of the benchmarks are taken from [38]. Iterative benchmarks, such as Skeleton, were set to execute 15 iterations. With non-iterative benchmarks, we assume that data layouts can be freely chosen, while iterative benchmarks are assumed to have the input and output arrays in canonical layout.

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 [38].
Convolve	Convolve a 1000x1000 monochromatic image with a 5x5 Gaussian filter.
Life	Compute Conway's "Game of Life" on a 1000x1000 grid, more details in [38].
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 [4].

Table 2. A data-parallel benchmark suite used in evaluation.

5.3 Experimental Results

We implemented the localization analysis framework by taking dependence information (dependence polyhedra and h-transformations) from LooPo's [26] dependence tester and generating statement-wise affine transformations. Currently, after localization analysis, the parallel code is generated by hand using the optimizations presented in Sec. 4. We compiled generated C programs with gcc 3.4 using the "-O3" optimizing flag on an UltraSPARC III 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 with 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).

Benchmark	Comm. free?	Fully Localizable?	Partially Localizable?	k	q
Sum	Yes	N/A	N/A	0	0
Mv	No	No	Yes	0	0
Demosaic	No	Yes	N/A	0	2
Convolve	No	No	Yes	0	2
Life	No	Yes	N/A	1	1
2D Jacobi	No	Yes	N/A	1	1
Skeleton	No	Yes	N/A	1	1

Table 3. Localization analysis results of the benchmarks

Table 5.3 summarizes the localization analysis results for the benchmarks. For all benchmarks examined, there at least exists a computation allocation and data mapping that partially localizes data accesses.

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

1. Computation allocation based on the "owner-computes" rule and canonical data layout: referred to as *canon:owner_computes*.
2. Iteration space divided into P chunks along the outermost parallel loop and canonical data layout: referred to as *canon:static_chunk*.
3. Non-canonical data layout with padding and strip-mining, which corresponds to block-cyclic distribution, and computation allocation based on the "owner-computes" rule: referred to as *noncanon:block_cyclic*.
4. Non-canonical data layout with padding, strip-mining and permutation, corresponding to block distribution along one dimension; computation allocation based on the "owner-computes" rule: referred to as *noncanon:block*.

Fig. 9 shows speedups with the different computation allocation and data mapping schemes. A star is used to indicate the transformation chosen by our compile-time framework for each benchmark. As illustrated in Fig. 9, the best transformation always employs non-canonical data layout and is always predicted by our framework. Except for Sum, each benchmark is completely or partially localizable but not communication-free; thus the generated code takes a block-distributed data view to reduce communication.

We can observe significant performance improvement with the optimization. On average, the best transformation outperforms *canon:owner_computes* by 45.3% and *canon:static_chunk* by 16.3%. For iterative benchmarks including Life, 2D Jacobi and Skeleton, this improvement is more significant, up to 130% compared with *canon:owner_computes*. 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. 10 shows normalized on-chip network link utilization with different mapping schemes. On average, on-chip network traffic is reduced by 81.1% from *canon:owner_computes*. The most significant link utilization reduction is with Skeleton, which is 97.6% from *canon:owner_computes*. The results in Fig. 10 have implications beyond performance. Power has be-

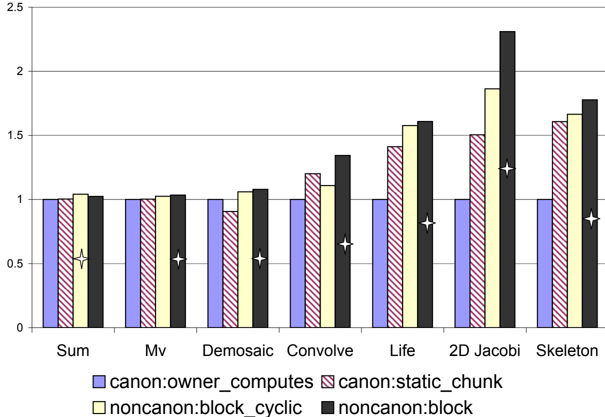


Figure 9. Speedups with different mapping schemes.

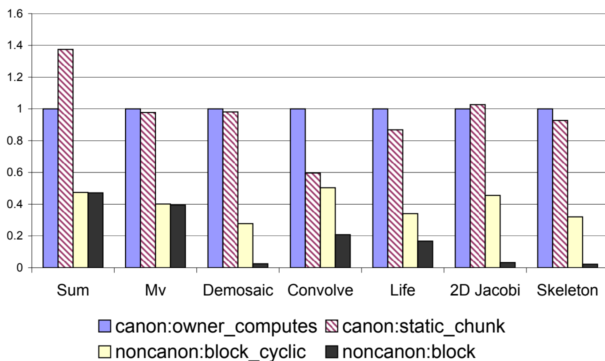


Figure 10. Normalized link utilization of on-chip network with different mapping schemes.

come 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 [27]. Several hardware/software approaches have been proposed to reduce the energy consumption from the on-chip network. [36, 10, 25] Link utilization reduction illustrated in Fig. 10 essentially translates to reduction in dynamic power dissipation over the on-chip network.

6. 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 [30, 5]. 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 [24] 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. [20] 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. [18, 19, 20] and Leung and Zahorjan [24] presented a global approach to this problem; of these, [18] considered dynamic layouts. The motivating context and our approach to solution are fundamentally different from the above efforts.

Chatterjee et al. [6] 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 is based on the polyhedral framework and non-canonical data layouts.

Barua et al. [2] 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. [2], So et al. [37] proposed to use custom data layouts to improve memory parallelism of FPGA-based platforms. In comparison to the above work, our study focuses on exploitation of distance locality instead of memory parallelism.

Rivera and Tseng [34, 35] 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. [8, 7, 9] 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. First, our objective is to improve the performance of data-parallel programs on tiled chip multiprocessors, while previous studies were mainly for performance improvement on uniprocessors. Second, 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. Finally, our approach considers arrays of different dimensionality while Chatterjee et al. focused on 2D arrays.

Anderson et al. [1] 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. [1] 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. The second phase of our code gener-

ation framework shares some similarities with their approach, but the first phase that performs localization analysis is very different.

7. Conclusion and Future Work

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

References

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

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

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

[4] 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 Trans. Softw. Eng.*, 26(3):197–211, 2000.

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

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

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

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

[9] S. Chatterjee and S. Sen. Cache-efficient matrix transposition. In *HPCA'00*.

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

[11] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO'06*.

[12] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[14] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. FMI, University of Passau, 2004. Habilitation Thesis.

[15] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible cmp cache sharing. In *ICS'05*.

[16] Intel Corporation. From a few cores to many: A tera-scale computing research overview. `ftp://download.intel.com/research/platform/terascale/terascale_overview%_paper.pdf`.

[17] R. N. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[18] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguade. Static and dynamic locality optimizations using integer linear programming. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):922–941, 2001.

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

[20] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):115–135, 1999.

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

[22] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[23] C. Lengauer. Loop parallelization in the polytope model. In *CONCUR'93*.

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

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

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

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

[28] 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*, 33(4):92–99, 2005.

[29] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. In *ISCA'07*.

[30] 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*.

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

[32] Peakstream. `http://www.peakstreaminc.com`.

[33] RapidMind. `http://www.rapidmind.net`.

[34] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *PLDI'98*.

[35] G. Rivera and C.-W. Tseng. Eliminating conflict misses for high performance architectures. In *ICS'98*.

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

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

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

[39] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[40] Virtutech AB. Simics full system simulator. `http://www.simics.com`.