

Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors



**Qingda Lu¹, Christophe Alias², Uday Bondhugula¹, Thomas Henretty¹,
Sriram Krishnamoorthy³, J. Ramanujam⁴, Atanas Rountev¹,
P. Sadayappan¹, Yongjian Chen⁵, Haibo Lin⁶, Tin-Fook Ngai⁵**

¹The Ohio State University, ²ENS Lyon – INRIA,

³Pacific Northwest National Lab, ⁴Louisiana State University,

⁵Intel Corp., ⁶IBM China Research Lab

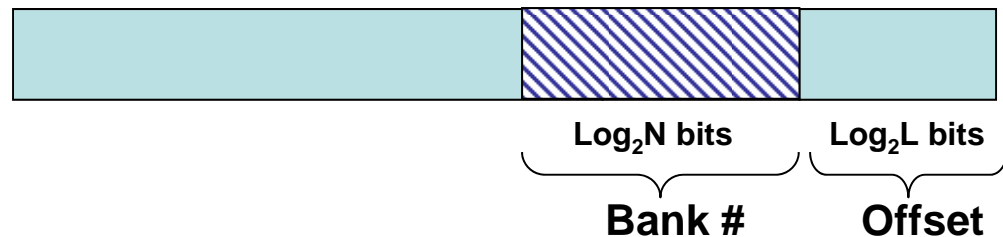
The Hardware Trends

- Trend of fabrication technology
 - Gate delay decreases
 - Wire delay increases
- Trend of chip-multiprocessors (CMPs)
 - More processors
 - Larger caches
 - Private L1 cache
 - Shared last-level cache to reduce off-chip accesses
 - We focus on L2 cache

Non-Uniform Cache Architecture

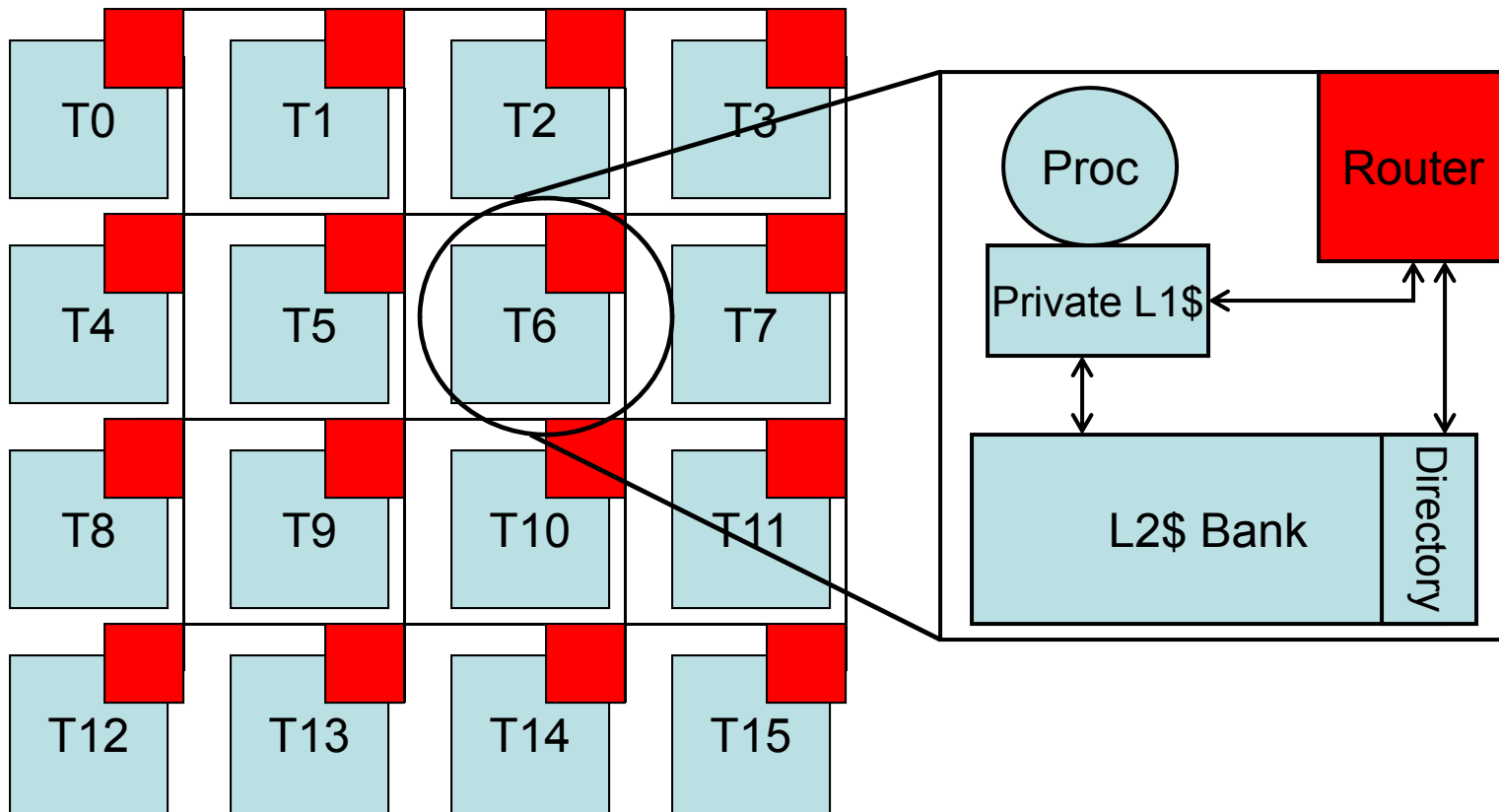
- The problem: Large caches have long latencies
- The Solution
 - Employ a banked L2 cache organization with non-uniform latencies
- NUCA designs
 - Static NUCA: Mapping of data into banks is predetermined based on the block index.

Physical Address



- Dynamic NUCA: Migrating data among banks. Hard to implement due to overheads and design challenges.

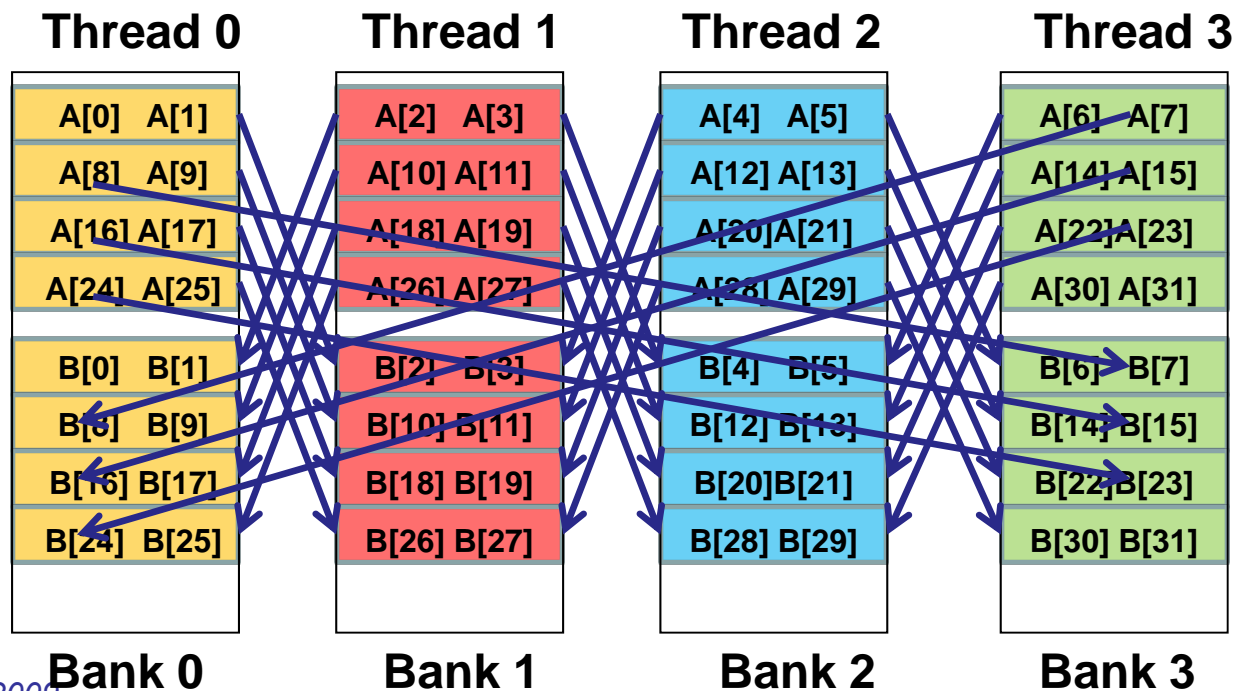
A Tiled NUCA-CMP



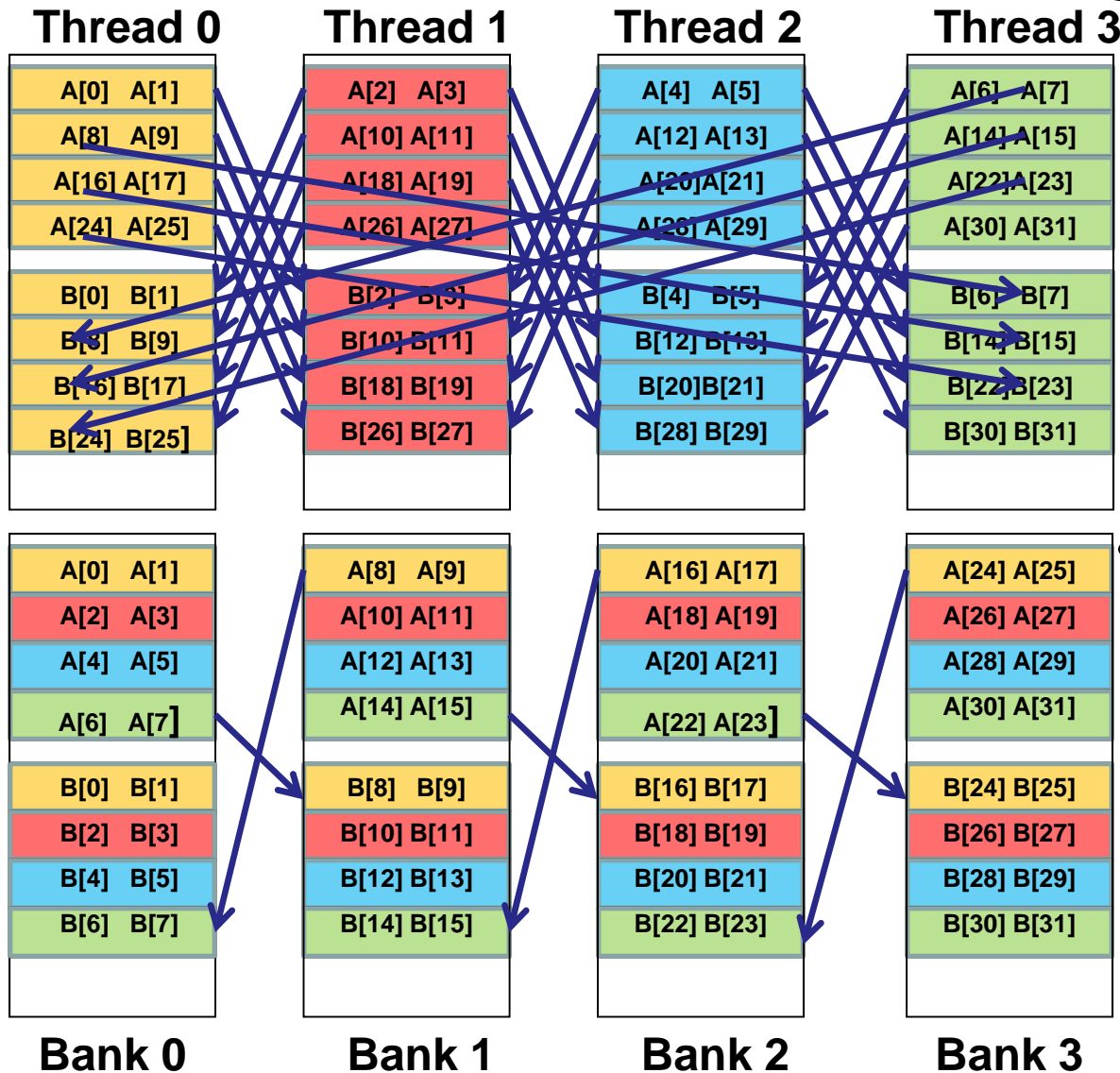
Motivating Example: Parallelizing 1D Jacobi on a 4-tile CMP

- 1-D Jacobi Code:


```
while (condition) {
  for (i = 1; i < N-1; i++)
    B[i] = A[i-1] + A[i] + A[i+1];
  for (i = 1; i < N-1; i++)
    A[i] = B[i];
}
```
- If we use the standard linear array layout and parallelize the program with the “owner-computes” rule, communication volume is roughly $2N$ cache lines in every outer iteration.



Motivating Example: Parallelizing 1D Jacobi on a 4-tile CMP



• If we divide the iteration space into four contiguous partitions and rearrange the data layout, only 6 remote cache lines are requested in every outer iteration.

The Approach

- A general framework for integrated data layout transformation and loop transformations for enhancing data locality on NUCA CMP's
- Formalism: polyhedral model
- Two main steps:
 - **Localization analysis**: search for an affine mapping of iteration spaces and data spaces such that no iteration accesses any data that is beyond a bounded distance in the target space
 - **Code generation**: generate efficient indexing code using CLooG by differentiating the “bulk” scenario from the “boundary” case via linear constraints.

Polyhedral Model

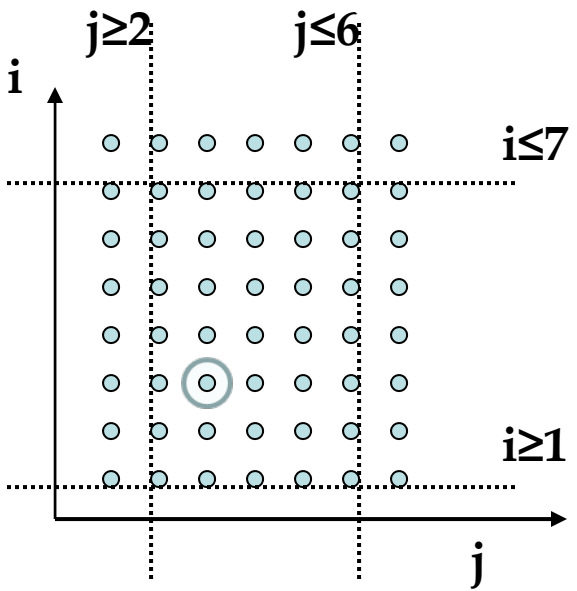
- An algebraic framework for representing affine programs – statement domains, dependences, array access functions – and affine program transformations
- Regular affine programs
 - Dense arrays
 - Loop bounds – affine functions of outer loop variables, constants and program parameters
 - Array access functions - affine functions of surrounding loop variables, constants and program parameters

Polyhedral Model

```

for (i=1; i<=7; i++)
  for (j=2; j<=6; j++)
    S1: a[i][j] = a[j][i] + a[i][j-1];
  
```

$$F_{S1,a}(\vec{x}_{S1}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$



$$\vec{x}_{S1} = \begin{pmatrix} i \\ j \end{pmatrix}$$

$$D_{S1}(\vec{x}_{S1}) = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 7 \\ 0 & 1 & -2 \\ 0 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \vec{0}$$

Computation Allocation and Data Mapping

- Computation allocation:
 - Original iteration space => Integer vector representing the virtual processor space
 - A one-dimensional affine transform (π) for a statement S is given by

$$\pi_S(\vec{x}_S) = C_S \cdot \begin{bmatrix} \vec{x}_S \\ \mathbf{n} \\ \mathbf{1} \end{bmatrix}$$

- Data Mapping:
 - Original data space => Integer vector representing the virtual processor space
 - A one-dimensional affine transform (ψ) for an array A is given by

$$\psi_A(\vec{X}_A) = G_A \cdot \begin{bmatrix} \vec{x}_A \\ \mathbf{n} \\ \mathbf{1} \end{bmatrix}$$

Localized Computation Allocation and Data Mapping

- Definition: 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 $F_{S,A}$,
 $\forall \vec{i}, D_S(\vec{i}) \geq 0 \Rightarrow \left| \pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) \right| \leq q$, 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 $F_{S,A}$ in a statement S , computation allocation π and data mapping ψ satisfy $\pi_S(\vec{i}) = \psi_A(F_{S,A}(\vec{i}))$

Localization Analysis

- **Step 1: Group Interrelated Statements/Arrays**
 - Form a bipartite graph: vertices corresponds to statements / arrays and edges connect each statement vertex to its referenced arrays.
 - Find the connected components in the bipartite graph.
- **Step 2: Find Localized Computation Allocation / Data Mapping for each connected component**
 - Formulate the problem as finding an affine computation allocation π and an affine data mapping ψ that satisfy $|\pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i}))| \leq q$ for every array reference $F_{S,A}$
 - Translate the problem to a linear programming problem that minimizes q .

Localization Analysis Algorithm

Require: Array access functions are rewritten to access **byte arrays**

$C = \emptyset$

for each array reference $F_{S,A}$ **do**

Obtain new constraints: under $\vec{i} \in D_S$, $\pi_S(\vec{i}) - \psi_A(F_{S,A}(\vec{i})) + q \geq 0$
and $\psi_A(F_{S,A}(\vec{i})) - \pi_S(\vec{i}) + q \geq 0$

Apply Farkas Lemma to new constraints to obtain linear constraints;
eliminate all Farkas multipliers

Add linear inequalities from the previous step into C

Add objective function (min q)

Solve the resulting linear programming problem with constraints in C

if ψ and π are found **then** return π , ψ , and q

else return “not localizable”

Data Layout Transformation

- Strip-mining

- An array dimension $N \rightarrow$ two virtual dimensions $(\left\lceil \frac{N}{d} \right\rceil, d)$
- Array reference array reference $[...] [i] [...] \rightarrow [...] [i / d] [i \bmod d] [...]$

- Permutation

- Array $A(\dots, N1, \dots, N2, \dots) \rightarrow A'(\dots, N2, \dots, N1, \dots)$
- Array reference $A[...] [i1] [...] [i2] [...] \rightarrow A[...] [i1] [...] [i2] [...]$.

Data Layout Transformation (Cont.)

A[0] A[1]
A[8] A[9]
A[16] A[17]
A[24] A[25]
B[0] B[1]
B[8] B[9]
B[16] B[17]
B[24] B[25]

A[2] A[3]
A[10] A[11]
A[18] A[19]
A[26] A[27]
B[2] B[3]
B[10] B[11]
B[18] B[19]
B[26] B[27]

A[4] A[5]
A[12] A[13]
A[20] A[21]
A[28] A[29]
B[4] B[5]
B[12] B[13]
B[20] B[21]
B[28] B[29]

A[6] A[7]
A[14] A[15]
A[22] A[23]
A[30] A[31]
B[6] B[7]
B[14] B[15]
B[22] B[23]
B[30] B[31]

A[0] A[1]
A[2] A[3]
A[4] A[5]
A[6] A[7]
B[0] B[1]
B[2] B[3]
B[4] B[5]
B[6] B[7]

A[8] A[9]
A[10] A[11]
A[12] A[13]
A[14] A[15]
B[8] B[9]
B[10] B[11]
B[12] B[13]
B[14] B[15]

A[16] A[17]
A[18] A[19]
A[20] A[21]
A[22] A[23]
B[16] B[17]
B[18] B[19]
B[20] B[21]
B[22] B[23]

A[24] A[25]
A[26] A[27]
A[28] A[29]
A[30] A[31]
B[24] B[25]
B[26] B[27]
B[28] B[29]
B[30] B[31]

Bank 0

Bank 1

Bank 2

Bank 3



The 1D jacobi example:
Combination of strip-mining and permutation

Data Layout Transformation(Cont.)

- Padding:
 - **Inter-array padding**: keep the base addresses of arrays aligned to a tile specified by the data mapping function
 - **Intra-array padding**: align elements inside an array with “holes” to make a strip-mined dimension divisible by its sub-dimensions

Data Layout Transformation (Cont.)

With localized computation allocation and data mapping:

- If we find communication-free allocation with all arrays share the same stride along the fastest varying dimension, only padding is applied
- Otherwise we create a blocked view of n-dimensional array A along dimension k with data layout transformations
 - If $K=1$, we have data layout transformation $\sigma(i_n, i_{n-1}, \dots, i_2, i_1) = (i_n, i_{n-1}, \dots, i_2, (i_1 \bmod (N_1/P))/L, i_1/(N_1/P), i_1 \bmod L)$, L is cache line size in elements and P is processor number.
 - If $K>1$, we have data layout transformation, $\sigma(i_n, i_{n-1}, \dots, i_2, i_1) = (i_n, \dots, i_1/L, i_k \bmod (N_k/P), \dots, i_2, i_k/(N_k/P), i_1 \bmod L)$.

Code Generation

- It is challenging to generate efficient code
 - Replacing array reference $A[u(i)]$ with $A'[\sigma(u(i))]$ produces very inefficient code
- We iterate directly in the data space of the transformed array.
 - In the polyhedral model, we specify an iteration domain D and an affine scheduling function for each statement.
 - Efficient code is then generated by CLooG that implements the Quilleré-Rajopadhye-Wilde algorithm.

Code Generation (Cont.)

Key techniques used in generating code accessing layout transformed arrays

- While σ is not affine, σ^{-1} is. We replace constraints $j_k = \sigma(u_k(i))$ by $\sigma^{-1}(j_k) = i$.
- We separate the domain D into two sub-domains D_{steady} and $D_{boundary}$ and generate code for two domains.
 - D_{steady} covers the majority of D by simplifying constraints such as $(i-u) \bmod L$ to $(i \bmod L) - u$
 - Boundary cases are automatically handled in $D_{boundary} = D - D_{steady}$

Experimental Setup

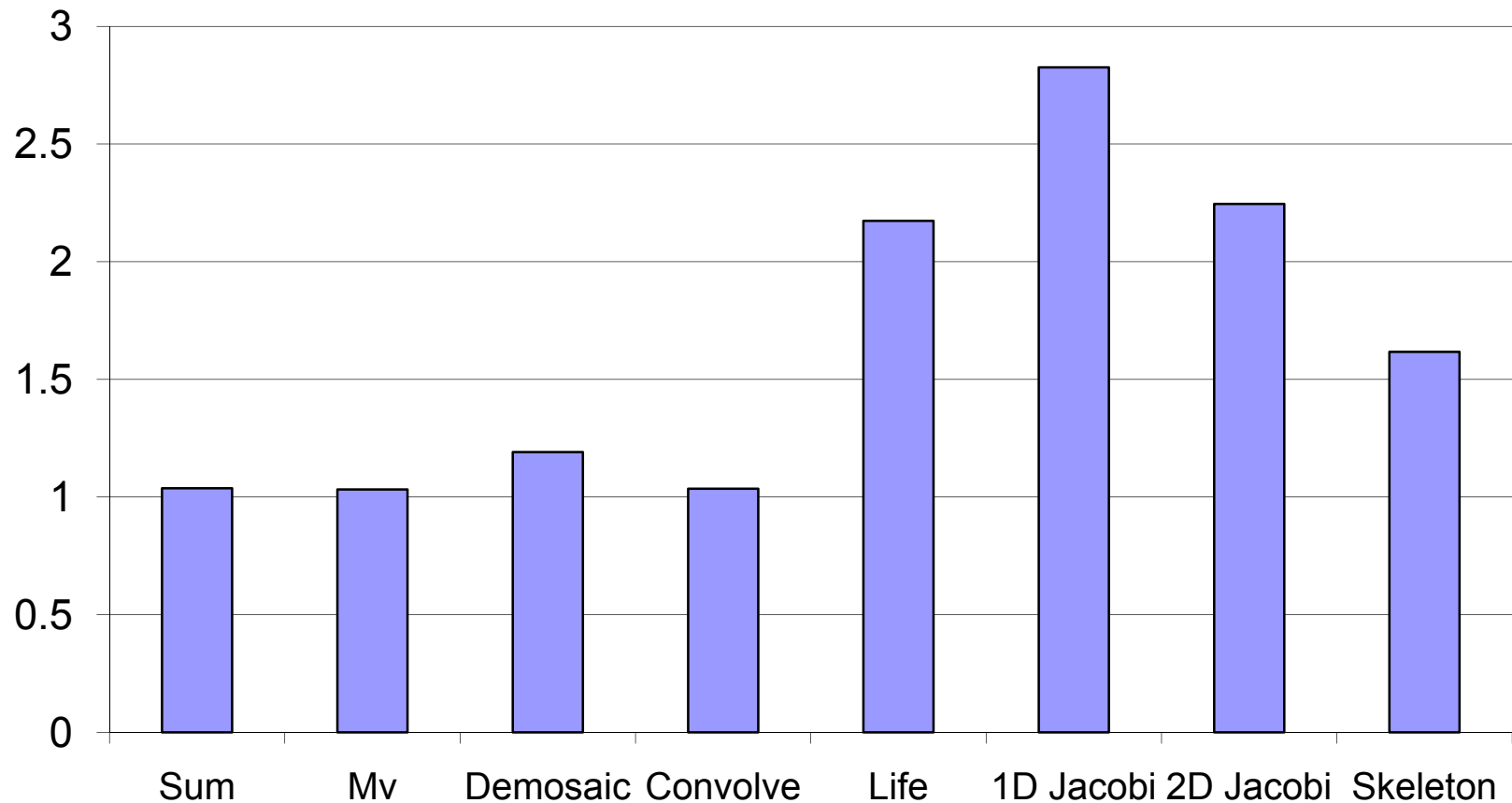
- With the Virtutech Simics full-system simulator extended with a timing infrastructure GEMS, we simulated a tiled NUCA-CMP:

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

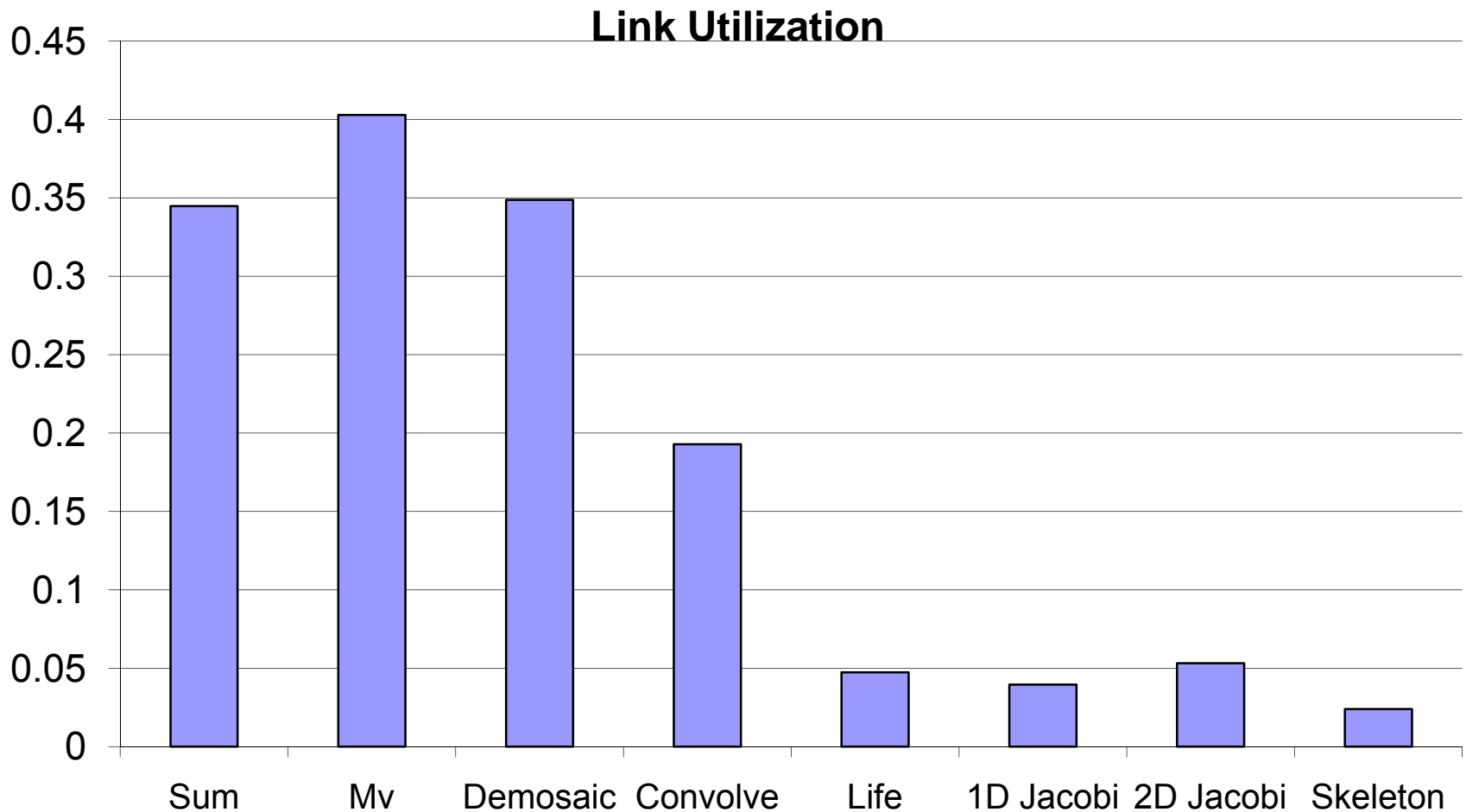
- We evaluated our approach with a set of data-parallel benchmarks
 - Focus of the approach is not on finding parallelism

Data Locality Optimization for NUCA-CMPs: The Results

Speedup

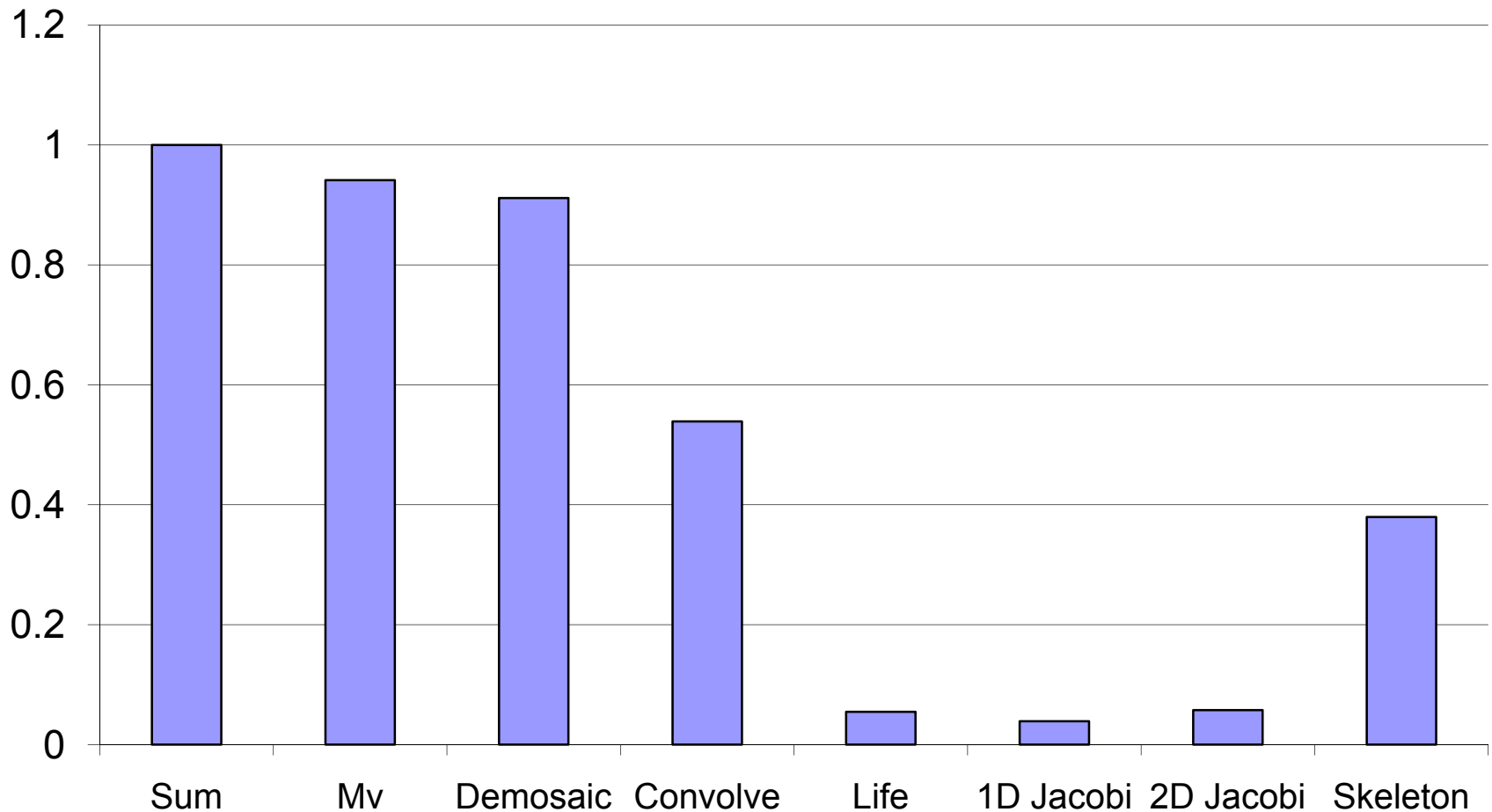


Data Locality Optimization for NUCA-CMPs: The Results



Data Locality Optimization for NUCA-CMPs: The Results

Remote L2 Access Number



Conclusion

- We believe that the approach developed in this study provides very general treatment of code generation for data layout transformation
 - analysis of data access affinities
 - automated generation of efficient code for the transformed layout