ABSTRACT

Recent research in embedded computing indicates that packing multiple processor cores on the same die is an effective way of utilizing the ever-increasing number of transistors. The advantage of placing multiple cores into a single die is that it reduces on-chip communication costs (in terms of both execution cycles and power consumption) between the processor cores that are traditionally very high in conventional high-performance parallel architectures (such as SMPs). However, on the negative side, this tighter integration exerts an even higher pressure on off-chip accesses to the memory system. This makes minimizing the number of off-chip accesses a critical optimization goal.

This paper discusses a compiler-based solution to this problem for the embedded applications that perform stencil computations. An important characteristic of this solution is that it distinguishes between the intra-processor data reuse and inter-processor data reuse. The first of these captures the data reuse that occurs across loop iterations assigned to the same processor, whereas the second one represents the data reuse that takes place across the loop iterations assigned to different processors. The proposed approach then optimizes inter-processor reuse by re-organizing the loop iterations of each processor carefully, considering how data elements are shared across processors. The goal is to ensure that the different processors access the shared data within a short period of time, so that the data can be captured in the on-chip memory space at the time of the reuse.

This paper also presents an evaluation of the proposed optimization and compares it to an alternate scheme that optimizes data locality for each processor in isolation. The results obtained by applying our implementation to eight loop-intensive benchmark codes from the embedded computing domain show that our approach improves over the mentioned alternate scheme by 15.6% on average.

Categories and Subject Descriptors
D.3.4 [Processors]: Compilers

General Terms
Performance

Keywords
Chip multiprocessors, stencil computation, data locality

1. INTRODUCTION

Recent research indicates that packing multiple processor cores on the same die is an effective way of utilizing ever-increasing number of transistors [12]. These chip multiprocessors have several advantages over complex single processor based architectures. On the hardware side, they are arguably easier to verify and validate, and since they are usually built from simple cores, they are more energy efficient as compared to sophisticated single processor based systems which are clocked at very high frequencies. On the software side, a chip multiprocessor gives the compiler writer the opportunity for exploiting both high-level (loop, thread) and low-level (ILP) parallelism. This support is very important for many embedded systems that execute loop-intensive image/video/speech processing applications [12].

An important advantage of placing multiple cores into a single die is that it reduces the communication costs (in terms of both execution cycles and power consumption) between the processor cores that are incurred in conventional high-performance parallel architectures (such as SMPs [9]). However, on the negative side, this tighter integration exerts an even higher pressure on off-chip accesses to the memory system. This is because in chip multiprocessors there are several cores that need to access the off-chip memory system, and they may have to contend for the same buses/pins to get there. Therefore, it is critical to reduce the number of off-chip memory accesses as much as possible, even if this causes an increase in on-chip communication activities among parallel processors.

Since early nineties compiler researchers focused on optimizations for cache locality and proposed several techniques along this direction. In the context of data caches, the proposed techniques include both loop transformations (e.g., iteration space tiling [22, 24] and loop permutation [2]) and data layout optimizations (e.g., dimension reindexing [14]). While one might think that these optimizations or some sort of combination of them can also be used in the context of chip multiprocessors, the problem is actually more complex than this simple view. This is because, optimizing the
Our concluding remarks and outlines the future work.

of the approach and compares it to previous work. Section 5 gives
the related work on stencil computations and chip multipro-
processors. The mathematical theory behind our approach is dis-
"tions by transforming the stencil codes for the best exploitation of
characteristics and improve the performance of stencil computa-
tions.
for (i=1; i<1024; i++)
for (j=1; j<1024; j++)
U[i][j]=U[i-1][j-1]+1;

Figure 2: Example loop and array accesses.

An array of dimension $n$ defines an array space $\mathcal{A}$, and each element in the array is specified using an index vector $\vec{A} = (a_1, a_2, ..., a_n)^T$. We assume that multi-dimensional arrays are stored in memory in a row-major fashion (in C and C++). We consider affine array access functions $f: \mathcal{A} \rightarrow \mathcal{A}$, $f(\vec{I}) = F\vec{I} + \vec{\zeta}$, where $F$ is an $n \times l$ matrix and $\vec{\zeta}$ is a $n$-dimensional constant vector. As an example, the two array accesses $U[i][j]$ and $U[i-1][j-1]$ in Figure 2 can be represented, respectively, as:

$$
(1 \ 0 \ 0)
\begin{pmatrix}
  i \\
  j
\end{pmatrix} +
(0 \ 0 \ 0)
\begin{pmatrix}
  0 \\
  0
\end{pmatrix} =
(1 \ 0 \ 0)
\begin{pmatrix}
  i \\
  j
\end{pmatrix} +
(-1 \ -1)
\begin{pmatrix}
  -1 \\
  -1
\end{pmatrix}.
$$

Linear loop transformations can be used to optimize a loop nest for various purposes, for example, improving cache locality. A linear loop transformation can be represented using an $l \times l$ non-singular matrix $T$ for a loop nest with $l$ loops [15, 23]. As a result of the loop transformation, each iteration (index vector) in the original iteration space is mapped to a distinct iteration in the new iteration space. If $\vec{I}$ is an iteration in the original iteration space, and is mapped to $\vec{I}'$ after the transformation represented by $T$, then the following equations must be satisfied:

$$
\vec{I}' = T\vec{I} \quad \text{and} \quad \vec{I} = T^{-1}\vec{I}'.
$$

After the transformation, the new access function, $f'(\vec{I}')$, and the original access function, $f(\vec{I})$, should access the same data element. That is: $f'(\vec{I}') = f(\vec{I})$. Considering Equation (1), we can obtain:

$$f'(\vec{I}') = f(\vec{I}) = f(T^{-1}\vec{I}') = FT^{-1}\vec{I}' + \vec{\zeta}.
$$

3.2 Algebraic Representation of Stencil Computations

A stencil can be represented by a stencil matrix, termed as $\mathcal{S}$, in which the columns are the relative positions of its neighbors assuming that the position of the seed element is $(0 \ 0 \cdots 0)^T$. $\mathcal{S}$ is an $n \times k$ matrix for an $n$-dimensional array and a stencil with $k$ neighbors. For example, the 5-point stencil in Figure 1(a) can be represented by a stencil $2 \times 4$ matrix $\mathcal{S}$, where:

$$\mathcal{S} =
\begin{pmatrix}
  1 & 0 & -1 & 0 \\
  0 & -1 & 0 & 1
\end{pmatrix}.
$$

Similarly, the stencil matrix for the 7-point stencil in Figure 1(b) is:

$$\mathcal{S} =
\begin{pmatrix}
  0 & 0 & 1 & 0 & 0 \\
  1 & 2 & 0 & 0 & -1 \\
  0 & 1 & -1 & -1
\end{pmatrix}.
$$

In a stencil, the neighbor elements can reside in different directions with respect to the seed element. For example, in Figure 1(a), there are four directions that hold neighbor elements of the seed, whereas in Figure 1(d) there are eight directions with neighbors. For a stencil on an $n$-dimensional array, there are a total of $2^n-1$ possible directions, and each direction can be represented by an $n$-entry direction vector. A direction vector is defined as a non-zero vector in which each entry’s absolute value is no more than one. A stencil can also be represented by a stencil direction matrix, termed as $D$, in which the columns represent the directions that hold neighbors. For example, the stencil direction matrix of the stencil in Figure 1(a) is

$$D =
\begin{pmatrix}
  1 & 0 & -1 & 0 \\
  0 & -1 & 0 & 1
\end{pmatrix},
$$

which is the same as its stencil matrix. The stencil direction matrix for the stencil in Figure 1(b) is

$$D =
\begin{pmatrix}
  0 & 1 & 0 \\
  1 & 0 & -1
\end{pmatrix}.
$$

The stencil direction matrix of a stencil can be obtained by removing the identical columns of $||\mathcal{S}||$, where $\mathcal{S}$ is the stencil matrix $(n \times k)$, and $||\mathcal{S}||$ is defined as:

$$||\mathcal{S}||_{(i,j)} =
\begin{cases}
  1 & \text{if } S(i,j) > 0; \\
  0 & \text{if } S(i,j) = 0; \\
  -1 & \text{if } S(i,j) < 0.
\end{cases}
$$

For example, by applying the $||\ ||$ operator to the stencil matrix of the stencil given in Figure 1(b), we obtain the following matrix:

$$
\begin{pmatrix}
  0 & 0 & 1 & 1 & 0 & 0 \\
  1 & 1 & 0 & 0 & -1 & -1
\end{pmatrix}.
$$

After removing the identical columns from the above matrix, we obtain the corresponding stencil direction matrix:

$$D =
\begin{pmatrix}
  0 & 1 & 0 \\
  1 & 0 & -1
\end{pmatrix}.
$$

3.3 Processor Representation and Array Assignment

We focus on a block distribution of arrays on a multi-dimensional processor space. In such a distribution, each processor updates a distinct subset (in the form of a block) of the array elements. The
position of each processor is specified using an \( n \times n \)-entry vector for an \( n \)-dimensional array. Figure 3(a) presents a program segment performing 5-point stencil computation. In this program, the outermost \( t \) loop controls the number of times the stencil computation needs to be repeated so that the results converge. At the end of each round, the results in array \( \text{temp}\mathbf{U} \) are copied back to \( \mathbf{U} \). The \((i, j)^T\) loop nest implements the actual 5-point stencil computation, and is the focus of our approach. Therefore, we will omit the outermost \( t \) loop and the array copying part (which is not a stencil computation) in the rest of our discussion. Figure 3(b) illustrates a \( 3 \times 3 \) block distribution of the array across nine processors for a 5-point stencil. These processors are represented in our framework using nine vectors: \((0, 0)^T, (0, 1)^T, (0, 2)^T, \ldots, (2, 2)^T\).

After array-to-processor assignment, each processor executes a subset of the original loop iterations. The loops to execute these iterations are the same for different processors, except that they have different lower and upper bounds. This is illustrated in Figure 3(c), which shows the part of the stencil computation in Figure 3(a) assigned to the processor identified with \((p_1, p_2)^T\), i.e., its local iteration space.

We extend the \( \text{abs}() \) function, which returns the absolute value of an integer, to the domain of vectors. That is:

\[
\mathbf{V}^T = \text{abs}(\mathbf{V}) \Rightarrow \forall j, 0 \leq j < n, \mathbf{V}^T(j) = \text{abs}(\mathbf{V}(j)).
\]

An \( n \times n \)-entry vector is called a unit vector if it is one of the columns of an \( n \times n \) identity matrix \( E \). We use \( E_j \) \((0 \leq j < n)\) to represent a unit vector that has a ‘1’ in its \( j \)th entry, and all ‘0’s in other entries. For example, we have \( E_0 = (1, 0, \ldots, 0)^T \) and \( E_{n-1} = (0, 0, \ldots, 1)^T \). Two processors \( P_1 \) and \( P_2 \) are said to be neighbors to each other if and only if \( \text{abs}(\mathbf{P}_1 - \mathbf{P}_2) \) is a unit vector. The vector \( \text{abs}(\mathbf{P}_2 - \mathbf{P}_1) \) is called the neighboring direction between \( P_1 \) and \( P_2 \). For example, in Figure 3, processors \((1, 1)^T\) and \((1, 2)^T\) are neighbors because \( \text{abs}((1, 1)^T - (1, 2)^T) = (0, 1)^T \), which is a unit vector. In comparison, processors \((1, 1)^T\) and \((2, 2)^T\) are not neighbors because \( \text{abs}((1, 1)^T - (2, 2)^T) = (1, 0)^T \), which is not a unit vector.

### 3.4 Inter-Processor Data Reuse

In stencil computations, neighboring processors share data at their boundaries. Figure 3(b) illustrates such data sharing. The shaded (gray) areas in Figure 3(b) represent the array elements that are shared by neighboring processors. For example, \( R_1 \) captures the array elements shared by processors \((0, 0)^T\) and \((0, 1)^T\), and \( R_2 \) indicates the array elements shared by processors \((1, 1)^T\) and \((2, 2)^T\). For processors that are not neighbors, they either do not share data (e.g., processor \((0, 0)^T\) and processor \((0, 2)^T\)), or they share a very small amount of data (e.g., processor \((0, 0)^T\) and processor \((1, 1)^T\) in a 9-point star stencil). Consequently, in our approach, we consider data sharing only between the neighboring processors.

While all neighboring processors share data in Figure 3(b), in the general case, whether two neighboring processors share data or not depends on the stencil direction matrix \( D \) as well. Figure 4(a) gives a 3-point stencil and Figure 4(b) shows its stencil direction matrix. Figure 4(c) illustrates the original data access pattern of an individual processor. In this data access pattern, array data is accessed row by row in the increasing order (from top to bottom), and the data within each row is accessed in the increasing order (from left to right). Figure 4(d) highlights the data sharing between processors in a \( 3 \times 3 \) block distribution. We can observe from this figure that there is no data sharing between the neighboring processors in the same column, but there is data sharing between the neighbor-
It is important to note that data sharing between processors does not necessarily lead to inter-processor data locality. Let us consider Figure 4(d) as an example. When processor \((0 1)^T\) accesses the data in the region indicated by \(R_1\), processor \((0 1)^T\) is accessing the data in the region indicated by \(R_2\). Since they do not access their shared data together (i.e., at the same time), there is little data locality in this access pattern (i.e., data locality is not fully exploited). On the other hand, in the data access patterns shown in Figure 4(e) and Figure 4(f), neighboring processors access their shared data together, and thus there is good inter-processor data locality in these two scenarios. Our objective is to transform the local iteration spaces of the processors so that their new data access pattern is similar to the ones shown in Figure 4(e) or Figure 4(f). We observe that the common characteristic between the access patterns of Figure 4(e) and Figure 4(f) is that the neighboring processors that share data proceed in the opposite directions when accessing each row and they proceed in the same direction when accessing each column. For example, in Figure 4(e), processor \((0 0)^T\) accesses each row from left to right, while processor \((1 0)^T\) accesses each row from right to left. Also, both these two processors access each column from top to bottom. Note that in this case, accessing each row means accessing along the direction identified by \(abs((0 1)^T - (0 0)^T)\) i.e., their neighboring direction. In general, for two neighboring processors to access their shared data together, they need to have \(opposite\) access directions when accessing data along their neighboring direction, and have the same access direction when accessing data along any other directions.

Let us now define the access directions in different dimensions for a given array access function \(f(\vec{I})\). We assume that each loop iterator is increased by one as we move from one iteration to another. The loops with non-unit steps can be transformed to unit-step loops using loop normalization [1]. The access direction vector \(\vec{Q}_k\) for the \(k\)th loop can be defined as:

\[
\vec{Q}_k = f(\vec{I} + E_k) - f(\vec{I}) = F(\vec{I} + E_k) + \vec{\zeta} - F(\vec{I}) - \vec{\zeta} = FE_k.
\]

That is, \(\vec{Q}_k\) is the \(k\)th column of the access matrix \(F\) of \(f(\vec{I})\). Therefore, the access matrix of \(f(\vec{I})\) is also the access direction matrix, in which the \(k\)th column is the access direction vector at the \(k\)th loop. From Equation (2), we can see that the new access direction matrix after the loop transformation represented by matrix \(T\) is \(T^{-1}F\).

Our approach is to find a suitable loop transformation for each processor so that we can have data reuse between the processors that share data. Our approach proceeds as follows. First, we identify all the neighboring processor pairs. After that, for each processor pair, we determine whether they share data or not using Equation (3). As has been discussed earlier, for two neighboring processors \(P_1\) and \(P_2\) that share data, they should have the opposite access directions along their neighboring direction \(abs(P_1 \pm P_2)\), and they should have the same access directions in other directions.

To formulate this requirement, we define a function \(\text{Expand}(\cdot)\) : \(n \times 1 \rightarrow n \times n\), which expands a direction vector into a matrix.

Mathematically, we can define \(\text{Expand}(\cdot)\) as follows:

\[
X = \text{Expand}(\vec{V}) \Rightarrow \begin{cases} 
1 & \text{if } i = j \text{ and } \vec{V}(i) = 0 \\
-1 & \text{if } i = j \text{ and } \vec{V}(i) \neq 0 \\
0 & \text{if } i \neq j
\end{cases} \quad (4)
\]

The function \(\text{Expand}(\cdot)\) can be used to express the requirement of the access directions of two neighboring processors that share data. The transformation matrices \(T_1\) (for processor \(P_1\)) and \(T_2\) (for processor \(P_2\)) should satisfy the following condition:

\[
FT_1^{-1} = \text{Expand}(\vec{P}_2 - \vec{P}_1)FT_2^{-1}. \tag{5}
\]

Since \(\text{Expand}(\vec{V})^{-1} = \text{Expand}(\vec{V})\), the above equation is equivalent to:

\[
FT_2^{-1} = \text{Expand}(\vec{P}_1 - \vec{P}_2)FT_1^{-1}. \tag{6}
\]

Equation (5) captures the requirement that, after these transformations, \(P_1\) and \(P_2\) should have opposite access directions along the neighboring direction between \(P_1\) and \(P_2\), and have the same access direction in all others.

Assume that there are \(V\) processors and \(W\) processor pairs that share data. Therefore, we have \(W\) equations in the form of Equation (5), and \(V\) unknowns (i.e., \(V\) transformation matrices to be determined) in these equations. After solving this set of equations, we can obtain the transformation matrices needed to transform the local iteration space of each processor so that the inter-processor data reuse can be exploited. Figure 5 gives a sketch of our compiler algorithm. Note that if one of the transformation matrix is known in Equation (5), this equation can be simplified as follows:

\[
AT = B, \tag{7}
\]

where \(A\) and \(B\) are known, and \(T\) is the unknown matrix. There exist several algorithms [3, 8] that can be used for solving \(AT = B\), and we can use any of these algorithms in our approach. In the algorithm shown in Figure 5, we maintain two sets of equations. The set \(\mathcal{L}\) contains all the equations that have two unknown matrices in them, while the set \(\mathcal{C}\) contains all the equations that have exactly one unknown matrix. At each step, we try to solve an equation which has only one unknown matrix. Once we obtain a solution for this equation, all the equations in which both the transformation matrices are known are removed from \(\mathcal{C}\), and all the equations with exactly one unknown matrix are moved from \(\mathcal{L}\) to \(\mathcal{C}\). When \(\mathcal{C}\) becomes empty and \(\mathcal{L}\) is not empty, we randomly select an equation from \(\mathcal{L}\), and set one of the unknown matrices to identity matrix \(E\) (i.e., the loop of the corresponding processor is not transformed). If an equation has no solution, we simply ignore that equation, in which case the inter-processor data reuse represented by this equation cannot be exploited. Assuming that the complexity of solving Equation (7) is \(Y\) and the number of equations is \(W\), the complexity of our algorithm in Figure 5 is \(\mathcal{W}Y\).

### 3.5 Impact on Intra-Processor Data Reuse

There are two types of intra-processor data reuses: self-reuse and group-reuse [22]. Self-reuse refers to the situation where the same array reference accesses adjacent data in successive loop iterations. Group-reuse refers to the situation where two array references access adjacent data in successive loop iterations. In this section, our goal is to show that transformation matrices obtained from our algorithm preserve all self-reuse in the original program and preserve the group-reuse in the most frequent cases.

Mathematically, self-reuse of an access function \(f(\vec{I}) = F\vec{I} + \vec{\zeta}\) can be defined as:

\[
f(\vec{I} + (0 \ 0 \cdots 1)^T) - f(\vec{I}) = (0 \ 0 \cdots \delta)^T, \tag{8}
\]
where \( \text{abs}(\delta) \) is a small integer number. The above equation can be represented in matrix/vector form as follows:
\[
F(0 0 \cdots 1)^T = (0 0 \cdots \delta)^T. \tag{9}
\]

The group-reuse between two access functions, \( f(I) = FT + \zeta \) and \( f'(I) = F'T + \zeta' \), can be defined as:
\[
f'(I + (0 0 \cdots 1)^T) - f(I) = (0 0 \cdots \delta)^T, \tag{10}
\]
where \( \text{abs}(\delta) \) is a small integer number. The different access functions in a stencil computation have the same access matrix (i.e., \( F' = F \)). Therefore, the above equation can be represented in a matrix/vector form as follows:
\[
F(0 0 \cdots 1)^T = (0 0 \cdots \delta)^T + (\zeta - \zeta'). \tag{11}
\]

**Lemma 1:** In Equation (5), if processor \( \tilde{P}_2 \) exhibits self-reuse after loop transformation \( T_2 \), then processor \( P_1 \) also exhibits self-reuse after loop transformation \( T_1 \).

**Sketch of the proof:** After the transformations, the new access matrix of \( \tilde{P}_2 \) is \( FT_1^{-1} \) and the new access matrix of \( P_1 \) is \( FT_2^{-1} \).

Mathematically, Lemma 1 is equivalent to:
\[
FT_2^{-1}(0 0 \cdots 1)^T = (0 0 \cdots \delta_2)^T \Rightarrow FT_1^{-1}(0 0 \cdots 1)^T = (0 0 \cdots \delta_1)^T,
\]
where both \( \text{abs}(\delta_1) \) and \( \text{abs}(\delta_2) \) are small integer numbers. Let \( X = \text{Expand}(\tilde{P}_2 - \tilde{P}_1) \), we obtain:
\[
FT_1^{-1}(0 0 \cdots 1)^T = X F T_2^{-1}(0 0 \cdots 1)^T = X (0 0 \cdots \delta_2)^T = (0 0 \cdots \pm \delta_2)^T.
\]

The last step can be inferred from the definition of \( \text{Expand}() \) given by Equation (4).

A processor \( \tilde{P} \) obtains its transformation \( T \) in two possible ways. One way is through line 6 in the algorithm in Figure 5. In this scenario, since the transformation matrix is the identity matrix \( E \), which means no transformation, self-reuse is clearly preserved. The other way is through line 13, in which \( T \) obtains its value by solving an equation in the form of Equation (5). By using induction, we can easily prove from Lemma 1 that the transformation matrices obtained from our algorithm preserve the self-reuses in the original program.

**Lemma 2:** In Equation (5), if the last column of \( F \) has only one non-zero entry and processor \( \tilde{P}_2 \) preserves group-reuse after loop transformation \( T_2 \), then processor \( P_1 \) also preserves group-reuse after transformation loop \( T_1 \).

**Sketch of the proof:** In mathematical terms, Lemma 2 is equivalent to:

**Condition 1:** The last column of \( F \) has only one non-zero entry and

**Condition 2:** \( F(0 0 \cdots 1)^T = (0 0 \cdots \delta_0)^T + (\zeta - \zeta') \) and

**Condition 3:** \( FT_2^{-1}(0 0 \cdots 1)^T = (0 0 \cdots \delta_2)^T + (\zeta - \zeta') \Rightarrow FT_1^{-1}(0 0 \cdots 1)^T = (0 0 \cdots \delta_1)^T + (\zeta - \zeta') \),

where \( \text{abs}(\delta_0), \text{abs}(\delta_1) \) and \( \text{abs}(\delta_2) \) are small integer numbers. Since \( F(0 0 \cdots 1)^T \) is the last column of \( F \), we can infer from the first two conditions that:
\[
F(0 0 \cdots 1)^T = (0 0 \cdots \Delta)^T, \text{ and } (\zeta - \zeta') = (0 0 \cdots \theta)^T.
\]

Therefore, the third condition can also be expressed as:
\[
FT_2^{-1}(0 0 \cdots 1)^T = (0 0 \cdots (\delta_2 + \theta))^T. \tag{12}
\]

Letting \( X = \text{Expand}(\tilde{P}_2 - \tilde{P}_1) \), we obtain:
\[
FT_1^{-1}(0 0 \cdots 1)^T = X F T_2^{-1}(0 0 \cdots 1)^T = X (0 0 \cdots (\delta_2 + \theta))^T = (0 0 \cdots \pm (\delta_2 + \theta))^T.
\]

The last vector above can be expressed as:
\[
\begin{cases}
(0 0 \cdots \delta_2)^T + (0 0 \cdots \theta)^T & \text{for the ‘+’ case;} \\
(0 0 \cdots (-\delta_2 - 2\theta))^T + (0 0 \cdots \theta)^T & \text{for the ‘-’ case.}
\end{cases}
\]

Since the distance between the data elements accessed by successive iterations in a stencil computation is small, the value of \( \text{abs}(\theta) \) is small. Therefore, \( \text{abs}(-\delta_2 - 2\theta) \) is also a small number. Considering that \( (\zeta - \zeta') = (0 0 \cdots \theta)^T \), we see that the transformation \( T_1 \) preserves the group-reuse. □

Similarly, we can prove from Lemma 2 that the transformation matrices obtained from our algorithm preserve the group-reuse in the original program if the last column of \( F \) has only one non-zero entry. Having only one non-zero entry in the last column of \( F \) requires that the index variable of the innermost loop appears in the access function only once. Although this seems restrictive, almost all the known stencil computations satisfy this requirement. Therefore, our algorithm preserves group-reuse in the most common cases.

### 3.6 Examples

In this section, we use two examples to illustrate how to use the mathematical framework in Section 3.4 to transform the local iteration space of each processor for exploiting inter-processor data reuse.

#### 3.6.1 Example 1

Let us consider the stencil computation presented in Figure 3. The stencil direction matrix \( D \) and the access direction matrix \( F \)
These two types of directions correspond to two direction vectors: share any data or not. There are two types of neighboring directions. In this figure, there are twelve neighboring processor pairs. For example, for processors \((0 0)^T\) and \((1 0)^T\), the two transformers corresponding to these two direction vectors, namely, \(T^{(0 1)}\) and \(T^{(1 0)}\), are:

\[
T^{(0 1)} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad T^{(1 0)} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.
\]

This means that all the neighboring processors share data. Figure 6(b) illustrates the data sharing patterns exhibited by the processors.

Figure 6(a) shows the original access pattern for each processor. Since the neighboring processors do not access the shared data at the same time, such an access pattern is not good as far as exploiting inter-processor data reuse is concerned. In the next step, we transform the program code for each processor so that the interprocessor data reuse can be exploited; i.e., the data reuse can be converted to data locality. We use \(T^{(0 1)}\) to represent the transformation matrix for processor \((p1)\) and \((p2)^T\). First, we apply \(Expand()\) to the direction vectors, namely, \((0 1)^T\) and \((1 0)^T\):

\[
X_1 = Expand((0 1)^T) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix};
X_2 = Expand((1 0)^T) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.
\]

After that, we build a set of equations containing twelve equations for the twelve neighboring processor pairs.

\[
\begin{align*}
FT^{-1}_{0,0} &= X_1 FT^{-1}_{0,1}; \\
FT^{-1}_{0,1} &= X_1 FT^{-1}_{0,0}; \\
FT^{-1}_{1,0} &= X_1 FT^{-1}_{1,1}; \\
FT^{-1}_{1,1} &= X_1 FT^{-1}_{1,0}; \\
FT^{-1}_{2,0} &= X_1 FT^{-1}_{2,1}; \\
FT^{-1}_{2,1} &= X_1 FT^{-1}_{2,0}; \\
FT^{-1}_{0,0} &= X_2 FT^{-1}_{0,1}; \\
FT^{-1}_{0,1} &= X_2 FT^{-1}_{0,0}; \\
FT^{-1}_{1,0} &= X_2 FT^{-1}_{1,1}; \\
FT^{-1}_{1,1} &= X_2 FT^{-1}_{1,0}; \\
FT^{-1}_{0,0} &= X_2 FT^{-1}_{0,1}; \\
FT^{-1}_{0,1} &= X_2 FT^{-1}_{0,0}; \\
FT^{-1}_{1,0} &= X_2 FT^{-1}_{1,1}; \\
FT^{-1}_{1,1} &= X_2 FT^{-1}_{1,0}; \\
FT^{-1}_{0,0} &= X_2 FT^{-1}_{0,1}; \\
FT^{-1}_{0,1} &= X_2 FT^{-1}_{0,0}; \\
FT^{-1}_{1,0} &= X_2 FT^{-1}_{1,1}; \\
FT^{-1}_{1,1} &= X_2 FT^{-1}_{1,0}; \\

\end{align*}
\]

For example, for processors \((0 0)^T\) and \((0 1)^T\), the open form of the equation is:

\[
\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} T^{-1} = \begin{pmatrix} 0 \end{pmatrix}.
\]

Using the algorithm given in Figure 5, we obtain a solution to the above set of equations:

\[
T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.
\]

These transformation matrices can be used to transform the local iteration spaces of processors. As an example, Figure 7(a) presents the original program code for processor \((1 1)^T\), and it is transformed, using \(T_{1,1}\), to the code given in Figure 7(b). After the transformations, we obtain the new data access pattern shown in Figure 6(b). Obviously, this new access pattern exploits inter-processor data reuse much better than the one shown in Figure 6(a). For example, in Figure 6(a), processors \((0 0)^T\) and \((0 1)^T\) do not access the shared array elements in \(R_1\) together, while in Figure 6(b) the array elements in \(R_1\) are accessed together by these two processors, in which case inter-processor data reuse is converted to inter-processor data locality. Similar observations can be made for all the other data regions shared by the neighboring processors.

3.6.2 Example 2

The code segment shown in Figure 8(a) is obtained by applying loop-tiling to the code segment given in Figure 3(a). This stencil computation is parallelized over four processors, and the default data access pattern is shown in Figure 8(b). The corresponding stencil direction matrix \(D\) and the access direction matrix \(F\) are:

\[
D = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad F = \begin{pmatrix} 8 & 0 & 1 & 0 \\ 0 & 8 & 0 & 1 \end{pmatrix}.
\]

There are three neighboring processor pairs, and all the neighboring direction vectors between them are of the form \((0 1)^T\). Consequently, applying Equation (3) to this directions vector, we obtain:

\[
(0 1) \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix} = (0 -1 0 1) \neq 0.
\]
for (i=1; i<384; i++)
for (j=1; j<384; j++)
for (x=1; x<8; x++)
for (y=1; y<8; y++)
temp[i][j][x][y] = (U[i][j][x][y] + U[i][j][x+1][y] + U[i+1][j][x][y] + U[i+1][j][x+1][y])/4;

Figure 8: (a) The original stencil code for Example 2. (b) Distribution across four processors and the default data access pattern. The shaded (gray) areas indicate the shared data, and the arrows represent access pattern. Captured in small squares are the access patterns used for visiting the elements inside each tile. (c) The new data access pattern after the loop transformations.

This means that all the neighboring processors share data. Figure 3(b) illustrates the data sharing patterns exhibited by the processors. As before, we use $T_{p1,p2}$ to represent the transformation matrix for processor identified by $(p_1, p_2)$. First, we apply $Expand()$ to the direction vector $(0, 1)^T$:

$$X = Expand((0, 1)^T) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$  

We then build a system of three equations for the three neighboring processor pairs. That is:

$$FT_{0,0}^{-1} = XFT_{0,0}^{-1}; \quad FT_{0,1}^{-1} = XFT_{0,1}^{-1}; \quad FT_{0,2}^{-1} = XFT_{0,2}^{-1};$$  

Finally, using the algorithm in Figure 5, we obtain a solution to the above set of equations:

$$T_{0,0} = T_{0,2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$  

$$T_{0,1} = T_{0,3} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$  

Figure 8(c) gives the new access pattern after applying these transformations to the local iteration spaces of our four processors. Again, the new (transformed) access patterns exploit the inter-processor data reuse much better than the original patterns given in Figure 8(b). These two examples show that one can exploit inter-processor data reuse by employing a customized loop transformation matrix for each processor in the system.

### Table 1: Important simulation parameters used in our experiments and their default values. Each processor has its own L1 cache and all processors share an L2 cache.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>8</td>
</tr>
<tr>
<td>L1 Size</td>
<td>8KB</td>
</tr>
<tr>
<td>L1 Line Size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>L1 Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>L1 Latency</td>
<td>1 cycle</td>
</tr>
<tr>
<td>L2 Size</td>
<td>2MB</td>
</tr>
<tr>
<td>L2 Line Size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Memory Access Latency</td>
<td>120 cycles</td>
</tr>
<tr>
<td>Bus Arbitration Delay</td>
<td>5 cycles</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>Strict LRU</td>
</tr>
</tbody>
</table>

4. EXPERIMENTAL RESULTS

To perform our experimental evaluation, we used the Simics toolset [19]. Simics is an instruction set simulator and operating system emulator. It allows simulation of multiple processors connected through an on-chip memory space, which can have multiple layers. We modified Simics to keep track of the behavior of shared and non-shared data separately. The default values of the important simulation parameters we used in our experiments are listed in Table 1. The code modifications required by our approach are automated within the SUIF infrastructure [20].

The benchmark codes used in this study are given in Table 2. The common characteristic of these codes is that they all perform some sort of stencil computation. The second column gives a description of each benchmark and the next one shows the amount of input data used for executing the benchmark. The fourth column gives the number of execution cycles for each benchmark when no locality optimization is applied. The goal of our approach is to reduce the execution cycles by minimizing the number of off-chip references. The execution cycle reductions given in the remainder of this section are normalized values with respect to the last column of Table 2.

Figure 9 gives the number of off-chip memory references for our benchmarks. The bar marked “Reused” correspond to the number of visits to the off-chip memory for a data element that has previously been on the on-chip memory space. In other words, it captures the number of references to the off-chip memory due to not being able to exploit inter-processor data reuse while the data is in
Table 2: Stencil applications used in our experiments.

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Brief Description</th>
<th>Input Size (KB)</th>
<th>Execution Cycles (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss-Seidel</td>
<td>Gauss-Seidel Computation</td>
<td>3731.4</td>
<td>386.9</td>
</tr>
<tr>
<td>Weather</td>
<td>Weather Prediction</td>
<td>5982.2</td>
<td>901.1</td>
</tr>
<tr>
<td>Edge</td>
<td>Edge Detection Algorithm</td>
<td>5418.5</td>
<td>513.2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Jacobi Iterative Solver</td>
<td>1545.0</td>
<td>155.7</td>
</tr>
<tr>
<td>VB 2.0</td>
<td>Vertex Blending</td>
<td>7816.4</td>
<td>995.1</td>
</tr>
<tr>
<td>Map 1.1</td>
<td>Cube Mapping</td>
<td>2998.9</td>
<td>372.3</td>
</tr>
<tr>
<td>RB-SOR</td>
<td>Red-Black Successive-Over-Relaxation</td>
<td>4115.6</td>
<td>664.8</td>
</tr>
<tr>
<td>TDer</td>
<td>Terrain Detection</td>
<td>6695.1</td>
<td>756.9</td>
</tr>
</tbody>
</table>

the on-chip memory space. The bar marked “Total”, on the other hand, gives the total number of off-chip memory references. We observe that, on an average, nearly 56% of the off-chip references are due to not being able to exploit the inter-processor data reuse, which indicates an approach that can convert these misses in the on-chip memory space to hits can be very effective in practice.

Figure 10 summarizes the savings achieved by our approach in three groups. Note that, before applying our approach that optimizes inter-processor reuse, we optimized intra-processor reuse for each processor independently. The first bar for each benchmark gives the percentage savings (reductions) in the off-chip memory references that belong to the “Reused” category. The second bar gives the percentage savings when all the off-chip references are considered. We see that the average saving in the “Reused” and “Total” categories are 77.6% and 42.4%, respectively. The last bar shows the reductions in execution cycles. We observe that our approach reduces the overall execution cycles by 22.6% on the average. Figure 11 gives similar results for the alternate scheme that optimizes locality for each processor in isolation (i.e., that optimizes for intra-processor locality only). The most striking observation from this figure is that the reductions in the “Reused” category is very low compared to our approach, and amounts to 27.1% when averaged over all benchmark codes in our experimental suite. This is mainly because this alternate approach does not consider the reuse of the data shared by multiple processors. However, when we look at the second bar for each benchmark (marked “Total”), we see that the average savings is about 32.2% when all benchmarks are considered. That is, although it is not very effective for the shared data, this approach is successful in converting the remaining misses into hits in the on-chip memory space. We also see from Figure 11 that the average reduction in execution cycles with this approach is around 8.2%. To sum up, by comparing Figures 10 and 11, one can see that exploiting locality for the shared data is very important for stencil type of applications.

We now change the default number of processors used in our experiments so far, and conduct a sensitivity analysis. Figure 12 plots the percentage improvements in execution cycles when our approach is used. For each application in this graph, each bar corresponds to a processor count (from 2 to 64). Recall that the default values used so far was 8. Maybe the most important conclusion one might draw from these results is that the effectiveness of our approach increases as we increase the number of processors. For example, while the average saving with 4 processors is around 14.2%, that with 32 processors is about 34.3%. This is because when the number of processors is increased, interprocessor communication (i.e., data sharing) also increases (i.e., some of the intra-processor computation – before we increase the processor count – needs nonlocal data after we increase the processor count). This in turn increases the importance of optimizing the locality behavior of the shared data. We do not present the results with the alternate scheme here in detail, but want to say that our approach consistently generated better results than the alternate scheme.

5. CONCLUSIONS AND FUTURE WORK

Minimizing the number of off-chip memory references is very important in embedded chip multiprocessors from both the performance and power perspectives. This paper proposes and evaluates a compiler-based solution to this problem. It primarily focuses on data shared across processors, and re-organizes loop iterations assigned to processors in a coordinated fashion so that the reuse distance to shared data is minimized. Our experiments with eight
Figure 12: Reduction in execution cycles with different processor counts.

benchmark codes from the embedded computing domain indicate significant reductions in off-chip memory accesses. We are in the process of evaluating the impact of other locality oriented optimizations on our inter-processor reuse representation. We are also working on developing a new code parallelization strategy that leads to better inter-processor reuse (in terms of both volume and sharing pattern).

6. REFERENCES