Polygonal Iteration Space Partitioning

Aniket Shivam[1], Alexandru Nicolau[1], Alexander V. Veidenbaum[1], Mario Mango Furnari[3], and Rosario Cammarota[2]

[1] University of California Irvine, Irvine, USA
[2] Qualcomm Research, San Diego, USA
[3] ICIB - National Council for Research, Pozzuoli, Italy

Abstract. This work presents a new set of loop transformations to expose and maximize data locality in loop-nests with non-uniform reuse patterns. The proposed set of transformations use the norms of the Polyhedral Model to represent loop-nests and then leverages such a representation to partition the iteration space into polygonally shaped partitions with maximum locality. However, the partitioning algorithm tends to produce partitions with complex geometry (shape) and with progressively smaller number of iterations, which, in practice, introduces much run-time overhead. This work also focuses on containing the number of partitions and properly manage their geometry at run-time, to contain unnecessary overhead. The proposed transformations also exposes loop level parallelism, by grouping together independent iterations, thus improving performance of both serial and parallel execution. In parallel execution a selective mapping of partitions to threads based on the type of reuse these partitions exhibit is proposed.
The proposed transformations show a consistent performance speedup on serial execution (up to 1.2x over Polly) and parallel execution (up to 3.17x over PLuTo) of some loop-nests.

Keywords: Polygonal Partitions, Shape and Size Independent Tiling, Temporal Locality, Polyhedral Model.

1 Introduction

Modern compilers, such as `LLVM`, `GNU GCC` and `Intel ICC` perform many loop transformations, such as tiling, strip-mining, fusion and interchange [10], to speedup program execution. Loop transformations, such as tiling [7], focus on grouping iterations (tiles) to improve data locality. Such transformations effectively speedup program execution when loop-nests exhibit uniform reuse distances between loop statements and across loop iterations. Tiles shape and size, determined based on the cache hierarchy organization, are usually constant and repeat during the loop execution to include all the iterations. For example, in a doubly-nested loop where iteration $I_{i,j}$ accesses array index $A_{i-1,j-1}$ and $A_{i+1,j+1}$, the formation of either square or rectangular tiles would help in improving locality. Tiling ensures that data remains in cache until $I_{i-1,j-1}$ and $I_{i+1,j+1}$ are computed.

However, tiling loop-nests with non-uniform reuse patterns still remains a challenge, due to the impossibility of defining a single set of dependency vectors which can govern a tile size and shape. For example, if iteration $I_{i,j}$ accesses array index $A_{i,j}$ and $A_{i+j,j}$, neither a single fixed-shape tile nor a symmetric tile can ensure improved cache data reuse during the whole execution of the loop-nest. The technique proposed by Meister et al. [9] for partitioning loops works irrespective of reuse pattern, i.e., it is not bound by constraint of shape and size of the tiles or partitions. The price of such a technique, however, is that the management of the boundary condition for the tiles introduces much instruction overhead, which the halt condition of the original partitioning algorithm does not account for.

This work proposes a new set of loop transformations to address the case of loop-nests with non-uniform reuse patterns, and to cope with the management of the execution of tiles of arbitrary shapes. Our proposed technique represents a loop-nest in the norms of the Polyhedral Model and then categorize iterations, i.e., create partitions based on the number of iterations that can linked by the reuse of their accessed data elements. In principle, the process could indiscriminately proceed until all the iterations in the loop belong to a partition. Alternatively, the compilation process may be set to halt at a predefined maximum number of partitions. However, the number of partitions has to be selected appropriately based on the characteristics of the loop-nest and the features of the target architecture to achieve maximum performance. We show that an optimal number of partitions can be determined per loop. Selecting more than the optimal number of partitions would introduce much overhead at run-time, whereas selecting less than the optimal number of partitions would miss a portion of exploitable locality and hence reducing speedup in both cases.

The proposed technique is implemented using the integration of source-to-source optimizer PLuTo[4] with PolyLib[5] library. The performance of the technique is compared against the combination of loop transformations already supported in Polyhedral Frameworks like Polly[6] and like PLuTo [4] (later compiled with `ICC`). Experimental results show a consistent speedup up to 1.2x w.r.t. Polly on serial execution and up to 3.17x w.r.t. PLuTo on parallel execution.

The rest of the paper is organized as follows: Section 2 presents our proposed set of loop transformations. Section 3 presents our experimental setup and results. Section 4 presents and comments on prior and related work. Finally, Section 5 summarizes our findings and presents our conclusive remarks.

2 Polygonal Iteration Space Partitioning

The proposed technique for generating the polygonal partitions of a loop-nest is presented in this section.

[4] PLuTo: http://pluto-compiler.sourceforge.net
[5] PolyLib: http://icps.u-strasbg.fr/~loechner/polylib/
[6] Polly (LLVM Plugin): http://polly.llvm.org

2.1 Determining Reuse using the Polyhedral Model

With the polyhedral representation of a nest of loops, a set of mathematical equations can be derived for identifying the data accessed by the references in a statement. For each instance of a statement in the body of a loop-nest, an iteration vector $\boldsymbol{I}$ is defined. For instance, if the enclosed statement accesses the data at a particular position of a multi-dimensional array $\mathbf{A}$, the exact location of the data ($A(\boldsymbol{I})$) can be calculated as: $A(\boldsymbol{I}) = \mathbf{R} \times \boldsymbol{I} + \mathbf{r}$. The *reference matrix*, R, is based on the coefficient of the iteration variables in the subscript representing the data access in A. Whereas, the *offset vector*, r, represents the constant from the subscript. For a D-dimensional array A, with N being the depth of the loop-nest, R will be a $D \times N$ matrix and r will be a D-dimensional vector identifying an offset in each dimension. To provide a explanatory example, consider the following loop-nest:

```
for (i = -N; i <= N; i++)
    for (j = -N; j <= N; j++)
        X[i,j] = Y[i,i+j+3] * Y[i+j,j];
```

The reference Y[i,i+j+3] references a two dimensional array Y enclosed in a two dimensional loop-nest. Therefore, R will be a 2×2 matrix, $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. Each row represents the projection of the reference along each dimension of the array, i.e., the value of subscript in each dimension (i and i+j+3). The column represents the coefficient associated with each iteration variable (i and j) of the loop-nest. The offset vector r is a column vector, $\begin{pmatrix} 0 \\ 3 \end{pmatrix}$, representing the offset for reference along every dimension, i.e., the constants in the subscript. An iteration $\boldsymbol{I}$ can be substituted using a column vector (i j). Each reference to the array is an unique combination of (R, r). The pair is represented as Γ to locate the accessed data point by an iteration. Γ is a function which computes the *image* of the polyhedron. In the above loop-nest, the two references to the array Y are written as: $\Gamma_{i,i+j+3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \boldsymbol{I} + \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ and $\Gamma_{i+j,j} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \boldsymbol{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

Suppose, there is reuse of a data by two different references Γ_α and Γ_β in iterations I_α and I_β respectively. Then, the dependence between two iterations can be described using Equation 1.

$$\Gamma_\alpha = \Gamma_\beta \Leftrightarrow R_\alpha I_\alpha + r_\alpha = R_\beta I_\beta + r_\beta \tag{1}$$

Therefore using Equation 1, as suggested in [9], the temporal reuse relation or dependence relation, $\mathcal{T}$, between I_α and I_β can be formally represented by Equation 2.

$$R_\beta^{-1} R_\alpha I_\alpha + R_\beta^{-1}(r_\alpha - r_\beta) = I_\beta \Leftrightarrow T_{\alpha\beta} I_\alpha + t_{\alpha\beta} = I_\beta, \quad \text{iff } R \text{ is invertible.} \tag{2}$$

The reuse relation $\mathcal{T}$ is a combination of $(T_{\alpha\beta}, t_{\alpha\beta})$, where $T_{\alpha\beta} = R_\beta^{-1} R_\alpha$ and $t_{\alpha\beta} = R_\beta^{-1}(r_\alpha - r_\beta)$. Substituting a particular iteration in place of I_α yields another iteration (I_β) that reuses the same data. If and only if R is invertible, then T can be computed. Therefore, the *reference matrix* R needs to be an square matrix. This implies that it is critical for the application of this technique that

the dimensions of the involved array is same as the depth of the loop-nest. This reduces the applicability to the loops with references that generate an invertible *reference matrix* R and hence an invertible $\mathcal{T}$. However, using R and $\mathcal{T}$ makes it possible to determine if a data accessed by I_β using Γ_β is also accessed by I_α using Γ_α, $I_\alpha = T_{\alpha\beta}^{-1} I_\beta$ - $T_{\alpha\beta}^{-1} t_{\alpha\beta}$. Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$ for the loop-nest in the example is: $T = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix}$ and $t = \begin{pmatrix} -3 \\ 3 \end{pmatrix}$ using Equation (2). In the example above, to check if the data accessed by an iteration, say i = 2 and j = 1, using reference $\Gamma_{i,i+j+3}$, is also accessed by another iteration using the reference $\Gamma_{i+j,j}$. Substituting the iteration vector by (2,1) in Equation 2, $\begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} -3 \\ 3 \end{pmatrix}$, yields vector (-4,6). Therefore, it can be concluded that iterations (2,1) and (-4,6) have reuse.

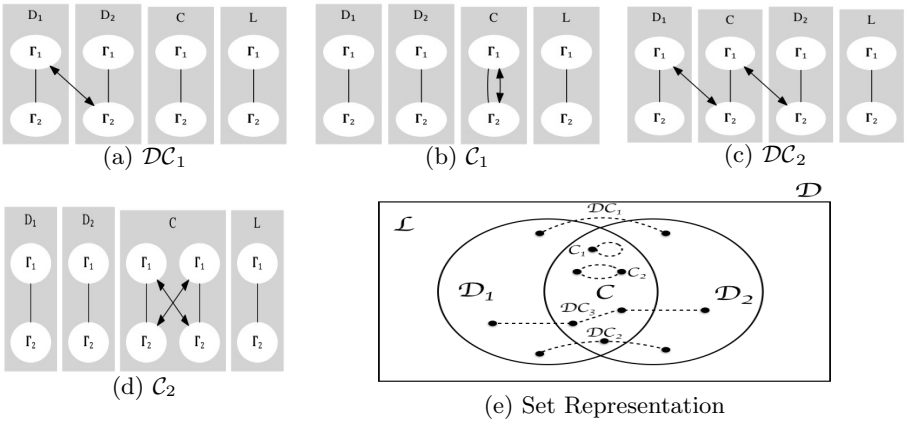
2.2 Partitioning Technique

The goal of our proposed technique is to identify and execute non-adjacent partitions of the iteration space in an order such that the data is reused in the cache. For an unoptimized version of the loop-nest, this data would have been flushed out of the cache before its reuse. These partitions are thereafter grouped based on the locality of the data their iterations access. Hence, all the partitions accessing the same set of data are aggregated. Assuming there are two references Γ_α and Γ_β to an array in a single statement in the loop-nest. The primary step is to partition the iteration space ($\mathcal{D}$) in three sets denoted by L, $\mathcal{P}_1$ and $\mathcal{P}_2$.

- $\mathcal{P}_1$ contain iterations that reference the data using Γ_α that another iteration in $\mathcal{D}$ accesses by Γ_β, i.e., these iterations have an *image* in $\mathcal{D}$ using relation $\mathcal{T}$.
- Iterations referencing the data using Γ_β that is also referenced by another iteration in $\mathcal{D}$ using Γ_α form the set $\mathcal{P}_2$. These iterations are the *images* of the iterations in $\mathcal{P}_1$. In other words, they have a *Pre-Image* in $\mathcal{D}$ ($Image(\mathcal{T}^{-1},\mathcal{D})$).
- The rest of the iterations in $\mathcal{D}$, i.e., the iterations that reference the data which is not referenced by another iteration are included in the partition denoted by $\mathcal{L}$. These iterations neither project nor they are projected in $\mathcal{D}$ using $\mathcal{T}$. Hence, $\mathcal{D} = \mathcal{P}_1 + \mathcal{P}_2 + \mathcal{L}$.

The sets $\mathcal{P}_1$ and $\mathcal{P}_2$ can be further categorized into three subsets named $\mathcal{C}$, $\mathcal{D}_1$ and $\mathcal{D}_2$, in addition to $\mathcal{L}$.

- $\mathcal{C}$: These iterations belong to both $\mathcal{P}_1$ and $\mathcal{P}_2$, i.e., $\mathcal{C} = \mathcal{P}_1 \cap \mathcal{P}_2$. Data accessed by these iterations using both the references (Γ_α and Γ_β) is also accessed by other iterations.
- $\mathcal{D}_1$: These iterations belong to $\mathcal{P}_1$ only, i.e., $\mathcal{D}_1 = \mathcal{P}_1 - \mathcal{C}$ or $\mathcal{D}_1 = \mathcal{P}_1 - \mathcal{P}_2$. The data accessed by Γ_α of these iterations is accessed by other iterations. Data accessed by Γ_β is not reused.
- $\mathcal{D}_2$: These iterations belong to $\mathcal{P}_2$ only, i.e., $\mathcal{D}_2 = \mathcal{P}_2 - \mathcal{C}$ or $\mathcal{D}_2 = \mathcal{P}_2 - \mathcal{P}_1$. Similarly, the data accessed by these iterations using Γ_β is reused, whereas data accessed by Γ_α remains unused.



(a) $\mathcal{DC}_1$ (b) $\mathcal{C}_1$ (c) $\mathcal{DC}_2$

(d) $\mathcal{C}_2$ (e) Set Representation

Fig. 1: Classification of iterations - formation of the sets $\mathcal{DC}_1$, $\mathcal{C}_1$, $\mathcal{DC}_2$, $\mathcal{C}_2$.

After categorizing the iterations based on the reuse of their accessed data, a further sub-categorization is performed such that each subset is executed in a specific order to improve the temporal locality. That is, iterations having reuse among them and forming smaller partitions ($\mathcal{DC}_k$ and $\mathcal{C}_k$) are linked together. Fig. 1 shows a graphical illustration of how iterations are categorized.

- $\mathcal{DC}_1$: $\mathcal{D}_1$ iterations that link to $\mathcal{D}_2$ iterations by $\mathcal{T}$, i.e., $\mathcal{DC}_1 = \mathcal{D}_1 \cap \mathcal{T}^{-1}(\mathcal{D}_2)$.
- $\mathcal{C}_1$: $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}$, i.e., $\mathcal{T}(\mathcal{C}_1) = \mathcal{T}^{-1}(\mathcal{C}_1)$.
- $\mathcal{DC}_2$: $\mathcal{D}_1$ iterations that link to $\mathcal{C}$ iterations that link to $\mathcal{D}_2$ iteration, i.e., $\mathcal{D}_1$ iterations that link to $\mathcal{D}_2$ iterations by $\mathcal{T}^2$, $\mathcal{DC}_2 = \mathcal{D}_1 \cap \mathcal{T}^{-1}(\mathcal{C}) \cap \mathcal{T}^{-2}(\mathcal{D}_2)$.
- $\mathcal{C}_2$: The remaining $\mathcal{C}$ iterations that form cyclic-link with one other iteration in $\mathcal{C}$, i.e., $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}^2$, $\mathcal{C}_2 = \mathcal{C} \cap \mathcal{T}^{-1}(\mathcal{C}) \cap \{I \in \mathcal{C} | T^2I + Tt + t = I\} - \mathcal{C}_1$.

After k repetitions of the previous steps:

- $\mathcal{DC}_k$: $\mathcal{D}_1$ iterations that link to chain of $k-1$ $\mathcal{C}$ iterations and at the end link to a $\mathcal{D}_2$ iteration by $\mathcal{T}^k$, i.e., $\mathcal{DC}_k = \{I \in \mathcal{D}_1 | Tt + t \in \mathcal{C}, T^2I + Tt + t \in \mathcal{C}, ..., T^kI + T^{k-1}t + ... + Tt + t \in \mathcal{DC}_2\}$.
- $\mathcal{C}_k$: The remaining $\mathcal{C}$ iterations that are linked to themselves by $\mathcal{T}^k$ forming a cyclic-link of k $\mathcal{C}$ iterations, i.e., $\mathcal{C}_k = \{I \in \mathcal{C} | Tt + t \in \mathcal{C}, T^2I + Tt + t \in \mathcal{C}, ..., T^kI + T^{k-1}t + ... + Tt + t = \mathcal{C}\} - \{\mathcal{C}_1 + ... + \mathcal{C}_{k-1}\}$.

These repetitive steps generate partitions based on the number of iterations that can linked by reuse of their accessed data elements. This partitioning technique requires a halting condition such that the number of steps of the algorithms, k, can be determined and so does determines the number of partitions that it creates. As mentioned in [9], the value of k can be chosen as: (a) If after the k^{th} repetition of the algorithm, the entire iteration space ($\mathcal{D}$) is completely partitioned. At this point $\mathcal{T}^k$ is an identity matrix, where $\mathcal{T}$ is represented as $\begin{pmatrix} T & t \\ 0..0 & 1 \end{pmatrix}$, and (b) If value of k is preset, the algorithm stops after the k repetitions and put the rest of the iterations in $\mathcal{C}_{k+1}$.

The partitions categorized as either $\mathcal{DC}_i$ or $\mathcal{C}_i$, where $1 \leq i \leq k$, are disjoint

partitions spread across the iteration space. Therefore, the partitions labeled as $\mathcal{DC}_i$ can be numbered based on the position of their containing iterations in the chain. In the $\mathcal{DC}_i$ partitions, the first partition containing only $\mathcal{D}_1$ iterations are labeled as $\mathcal{DC}_i^0$. The next $i-1$ partitions containing $\mathcal{C}$ iterations are labeled as $\mathcal{DC}_i^1$, $\mathcal{DC}_i^2$,...,$\mathcal{DC}_i^{i-2}$ and $\mathcal{DC}_i^{i-1}$. The last partition in the chain containing $\mathcal{D}_2$ iterations is labeled as $\mathcal{DC}_i^i$. The same naming paradigm is followed for $\mathcal{C}_i$ partitions. These i partitions are labeled as $\mathcal{C}_i^0$, $\mathcal{C}_i^1$,...,$\mathcal{C}_i^{i-2}$ and $\mathcal{C}_i^{i-1}$. The number of iterations in the partitions of similar type is always equal, since the iterations in the successive partitions are the *images* of the iterations in the previous partition.

2.3 Orchestrating Formation of the Partitions

Premature Halting. An indiscriminate application of the algorithm introduce overhead at run-time due to large number of small sized partitions, which is not considered in the halting conditions defined above. The increase in the number of partitions increases the control statement overhead in the restructured loop-nest. Therefore, in the partitions with very few iterations the gain in performance from better locality is overshadowed by the control overhead needed to manage such partitions.

We introduced an termination method for the algorithm so that the control statement overhead does not overshadow the speedup gained through maximizing locality, by predicting the minimum tile size. Specially in loop-nests where the longest chain of linked iteration is very long, i.e., T^k generates an identity matrix for a very high value of k, say k_{max}, it is critical to find an optimal value of $k < k_{max}$ to protect gained speedup from increasing control overhead. This is applicable to most loop nests with one dimensional non-uniform reuse pattern. Therefore, the algorithm is halted after partitioning for $\mathcal{T}^k$ and the remaining iterations form partition $\mathcal{C}_{k+1}$. From our experiments, it can be deduced that the algorithm must be halted if the number of iterations in newly generated partitions is below 25×25, .i.e., 625 iterations.[7]

Multi-Level Tiling. The partitions generated on each repetition of the technique are labeled as $\mathcal{DC}_i$ and $\mathcal{C}_i$, where $1 \leq i \leq k$. Partitions labeled as $\mathcal{DC}_i$ or $\mathcal{C}_i$ are set of separate and distantly located partitions of the iteration space. The execution order of these partitions influences the improvement in locality or improved cache hit-miss ratio at a certain cache level. A single partition targets the improvement in locality in the smallest cache with the least expensive data transfer cost, ideally L1 cache. The set of partitions in $\mathcal{DC}_i$ or $\mathcal{C}_i$ targets a larger cache that can be either L2 or L3 cache. This technique guarantees that for loops with non-uniform reuse pattern, the cost in terms of time spent in fetching data for reuse is reduced by making it available in closest possible cache level.

Locality on Parallel Execution of the Partitions. Loop-nests without any loop-carried dependences can be executed in parallel without any constraints.

[7] The number of integer points contained by a parameterized polyhedron is computed using the Ehrhart Polynomials as implemented in PolyLib.

But tiling such loops can improve the performance by improving locality so that the cost of data transfer is reduced. During parallel execution more fetches from private memory and lesser fetches from the shared memory improves the performance. Scheduling similar partitions (either a $\mathcal{DC}_i$ or $\mathcal{C}_i$, $1 \leq i \leq k$) on the same thread achieves the improvement in locality, since each thread finds the required data in private memory.

2.4 Multi-Reference Statements

We also extend the technique to statements with multiple references to the array and also to stencil computations that exhibit fixed pattern reuse in multiple directions. Every pair of temporal reuse relations lead to different partitions which on combining would generate a single partition. Reuse along multiple directions create a complex network of iterations linked by $\mathcal{T}$, therefore it is important to eliminate reuse relations such that iterations do not link to themselves by either $\mathcal{T}$ or $\mathcal{T}^2$. For example, the reuse vector $\boldsymbol{v}_{i,j-1}$ and $\boldsymbol{v}_{i,j+1}$ link themselves by $\mathcal{T}^2$. Therefore, one of them must be eliminated. Also, $\boldsymbol{v}_{i,j}$ must be eliminated since it links to itself by $\mathcal{T}$. One drawback of the original algorithm is that some pairs of reuse vectors produce partitions which consume the entire iteration space like $\boldsymbol{v}_{i,j+1}$ and $\boldsymbol{v}_{i+1,j}$. These pairs are eliminated. The aim is to find the '*pair*' (best set of two references) from all the references that generate the best possible partitions for maximizing locality.

Another heuristics to choose the *pair* is to select it based on the amount of reuse in the partitions that it creates. A ***reuse count function*** as shown in Equation 3 is used to predict the amount of reuse in the partitions can be appended in the original technique. This step involves choosing the best *pair* out of every set of two references - from those left after eliminating the redundant references - based on the amount of reuse that can be calculated from size of $\mathcal{DC}$s and $\mathcal{C}$s sets. When the algorithm is prematurely halted to reduce control statement overhead as described in the previous section, the residual iterations that form $\mathcal{C}_{k+1}$ are not counted towards the reuse.

$$Reuse(\Gamma_\alpha,\ \Gamma_\beta) = \sum_{i=1}^{k} i \times |\mathcal{DC}_i^0| + \sum_{i=1}^{k} i \times |\mathcal{C}_i^0| \tag{3}$$

This technique can also be extended to multiple statements enclosed in a loop-nest. Since, reuse of data from an array might occur between references spanning across multiple statements. These multiple references can be reduced to the best *pair* of references exploiting the maximum locality.

2.5 Code Generation paradigm

The code generation for these partitions begins by analyzing the polyhedron representation for each partition. This polyhedron representation contains the constraints (boundary hyperplanes) that define the affine boundaries for the partitions. These constraints are then scanned using the Fourier-Motzkin algorithm implemented in PolyLib and also using tools like CLooG [3]. CLooG generates code by scanning the polyhedrons and performs the union of distinct polyhedron to produce code with the least control statement overhead. The work in [9]

suggests a methodology to scan just the initial partition from each category, i.e., $\mathcal{DC}_i^0$ for the $\mathcal{DC}_i$ type partitions and $\mathcal{C}_i^0$ for $\mathcal{C}_i$ type partitions. The next steps is to derive the subscripts for the next iterations in the link using the reuse relation $\mathcal{T}$. Let, I, a column vector, represent the iterations in the $\mathcal{DC}_i^0$. The subscript for the iterations in the following partitions $\mathcal{DC}_i^1$, $\mathcal{DC}_i^2$,..., $\mathcal{DC}_i^i$ are derived from $\mathcal{T}(I)$, $\mathcal{T}^2(I)$,..., $\mathcal{T}^i(I)$ respectively. The locality is exposed in the successive statements since there is reuse between I and $\mathcal{T}(I)$ iteration, then in $\mathcal{T}(I)$ and $\mathcal{T}^2(I)$ iteration, etc. This methodology is efficient unless the value of k is high in which case it enormously expands the code size. The loop-nest for $\mathcal{DC}_i$ and $\mathcal{C}_i$ partitions encloses $i+1$ and i statements respectively. For some value of k, the code will have a minimum of k loop-nests for either $\mathcal{DC}$ or $\mathcal{C}$ type partitions and maximum of $k \times 2$ (k $\mathcal{DC}$ plus k $\mathcal{C}$) loop-nests. Each of them containing statements between 0 and k. For a higher value of k, a better solution is to find the union of the polyhedron representing a type of partitions ($\mathcal{DC}_i$ or $\mathcal{C}_i$) to generate code. Also, since each partition in $\mathcal{DC}_i$ or $\mathcal{C}_i$ type partitions contain equal iterations, they tend to form similar geometries. These geometries are recurring patterns and hence code generation for them requires slight modification in the boundary conditions of the control statements. These modification can be captured to form a basis for iterating through each partition of a particular type. Hence, reducing the total count of loop-nests in the code.

An important part of the speedup comes from re-partitioning the generated partitions to reduce boundary check overheads. This is performed by computing these partial partitions and scanning them so as compute multiple partitions in a single loop. The entire partitioning technique is shown in Algorithm 1.

For generating parallel code, we propose the use of OpenMP® Sections. It allows the *selective mapping* of a certain type of partitions onto a single thread. This improves the locality in each thread which in turn reduces the fetching of same data from shared memory on multiple threads. These sections are dynamically scheduled to achieve load balancing. However, the generation of a schedule for parallel execution of polygonal partitions of a loop-nest with non-uniform data dependence remains a challenge. Because if the execution of partitions as per the technique violates any data dependence, then modifying the execution order without violating dependence disrupts locality.

3 Experiments and Results

For evaluating our technique, we choose four cases in which the corresponding loop-nests exhibit different reuse patterns. These styles are: (a) **Two Dimensional Non-Uniform Reuse** in which the reuse pattern varies along both dimensions of a two dimensional iteration space; (b) **One Dimensional Non-Uniform Reuse** in which the reuse pattern varies along a single dimension; (c) **Symmetric or Uniform Reuse** in which the reuse is generally among neighboring iterations along a certain direction(s); (d) **Multiple References** in which loop-nests contains multiple references to an array in a single statement, e.g., as seen in benchmark suites like PolyBench[8].

[8] PolyBench/C 4.1: http://web.cse.ohio-state.edu/∼pouchet/software/polybench/

Algorithm 1 Polygonal Tile Generation

1: **Input**: A loop-nest with potential reuse on a dataset (array).
2: Eliminate set of references that link iterations to themselves by either $\mathcal{T}$ or $\mathcal{T}^2$. (Section 2.4)
3: **for** each set of two references (Γ_α, Γ_β) to the array **do**
4: Define the Reuse Relation $\mathcal{T}$ using the two references Γ_α and Γ_β.
5: Generate coarse partitions of the iteration space ($\mathcal{D}$): $\mathcal{P}_1$ ($Image(\mathcal{T},\mathcal{D})$), $\mathcal{P}_2$ ($Image(\mathcal{T}^{-1},\mathcal{D})$) and $\mathcal{L}$ (No reuse).
6: Categorize $\mathcal{P}_1$ and $\mathcal{P}_2$ into: $\mathcal{C}$=$\mathcal{P}_1 \cap \mathcal{P}_2$, $\mathcal{D}_1$=$\mathcal{P}_1 - \mathcal{P}_2$ and $\mathcal{D}_2$=$\mathcal{P}_2 - \mathcal{P}_1$.
7: **while** $\mathcal{D}$ is not completely partitioned **do**
8: Create partitions ($\mathcal{DC}_i$ and $\mathcal{C}_i$) that have iterations linked by relation $\mathcal{T}^i$.
9: **if** Iterations in the generated partitions is below 25×25 **then**
10: $k = i$ (Since the algorithm is halted, k is set to i.)
11: Put rest of the iterations in $\mathcal{C}_{k+1}$.
12: break
13: **end if**
14: Increment i.
15: **end while**
16: **end for**
17: Remove the set of references that produce a single partition which consume the entire iteration space. (Section 2.4)
18: On the remaining set of references, apply the **Reuse Count Formula** (Equation 3) to estimate the amount of reuse.
19: Choose the *pair* having the maximum reuse in their polygonal partitions for code generation.
20: Scan the polygonal partitions using the Fourier-Motzkin algorithm to generate the boundaries for the partitions.
21: Use the code generation tools like CLooG with modifications so as to generate array subscripts using the function $\mathcal{T}^i(I)$.
22: **Output**: Polygonally tiled iteration space that improves data locality.

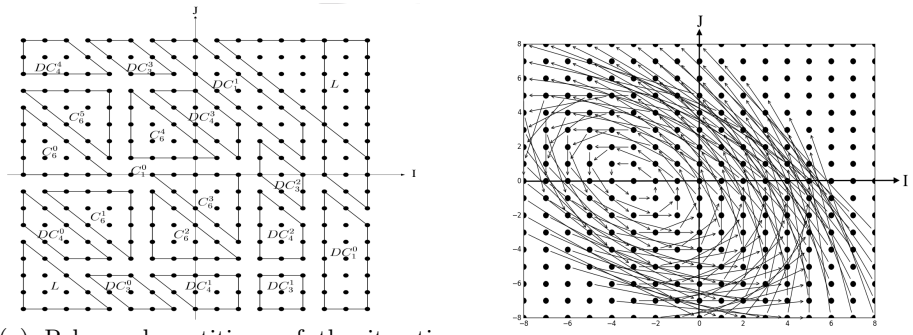
The compiled codes are analyzed for performance on Intel's Sandy-Bridge Core i7-2600 CPU @ 3.40GHz. The processor has 4 cores (8 threads) with 32 KB L1 I/D cache, 1024 KB L2 cache and 8 MB LLC. Hardware performance counters were analyzed for measuring performance metrics.

3.1 Case 1: Two Dimensional Non-Uniform Reuse Pattern

```
for (i = -N; i <= N; i++)
    for (j = -N; j<= N; j++)
        X[i,j] = Y[i,i+j+3] * Y[i+j,j];
```

Fig. 2: Case 1: Loop-Nest with Two Dimensional Non-Uniform Reuse

In the loop-nest shown in Fig. 2, the references to the array Y can be represented as: $\Gamma_{i,\ i+j+3} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 3 \end{pmatrix}$, $\Gamma_{i+j,\ j} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$ can be calculated using Equation 2, where $T = \begin{pmatrix} 0 & -1 \\ 1 & 1 \end{pmatrix}$ and $t = \begin{pmatrix} -3 \\ 3 \end{pmatrix}$. For this case k comes out to be 6, since T^6 is



(a) Polygonal partitions of the iteration space for Case 1 - $\mathcal{T}^6 = I$ (identity).

(b) RAR dependence in the loop-nest

Fig. 3: Partitions of the iteration space in Case 1.

an identity matrix. Therefore, the partitioning process would terminate after six repetitions of the core algorithm. The remaining iterations in $\mathcal{C}$ are placed in partition $\mathcal{C}_6$ as described in the technique. The graphical representation of the partitioned iteration space is shown in Figure 3a [9]. The partitioning algorithm generates a fixed number of partitions, which is independent of the input size. Hence, the partitions generated from this technique are scalable with the dataset size. Since, the maximum value of k is 6, it generates a small number of partitions which suggests that the control statement overhead will have negligible effect on performance. Therefore, there is no need to apply the halting condition described in Section 2.3 in this case. Hence, the maximum value must be chosen to obtain the finest partitions with the maximum reuse.

```
for (i = -N; i <= -4; i++) {
    for (j = MAX(-N+3,-i-N-3); j <= -i-N-1; j++) {
        X[i][j]           = Y[i][i+j+3] * Y[i+j][j];
        X[-j-3][i+j+3]    = Y[-j-3][i+3] * Y[i][i+j+3];
        X[-i-j-6][i+3]    = Y[-i-j-6][-j] * Y[-j-3][i+3];
        X[-i-6][-j]       = Y[-i-6][-i-j-3] * Y[-i-j-6][-j];
        X[j-3][-i-j-3]    = Y[j-3][-i-3] * Y[-i-6][-i-j-3];
    }
}
```

Fig. 4: Index calculation for $\mathcal{DC}_4$ using reuse relation($\mathcal{T}$).

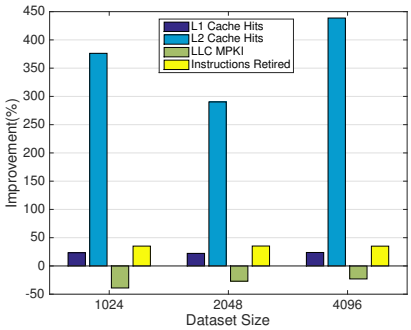


Fig. 5: Case 1: % Improvement in L1, L2, LLC and Instructions Retired Counters

The code shown in Fig. 4 presents the application of the function $\mathcal{T}^i(\mathrm{I})$ where $0 \leq i \leq 6$, as mentioned in Section 2.5, to compute array subscripts for disjoint but equivalent $\mathcal{DC}_4$ partitions. This optimization reduces the control statement overhead, as well as increases the temporal locality due to consecutive data accesses in subsequent iterations. Also, because of this there are less memory accesses and therefore there is a constant 35% de-

crease in instruction count. Fig. 5 shows the increase in cache hits.

The serial code optimized using the technique shows up to 1.19x speedup (Fig. 11a). For parallel execution, each type of partition is executed on a different thread using OpenMP® Sections so as to maximize data reuse on a core. On parallel execution the speedup is even higher (up to 3.17x) as shown in Fig. 11b due to the *selective mapping* of the partitions. Polly and PLuTo generate rectangular tiles for the given program, since both of them do not use the information from RAR dependence to optimize code for locality, unlike the proposed technique. Experimental results show scalability of performance with the input size because even though the number of partitions remains constant, the size of the partitions scales with the input size.

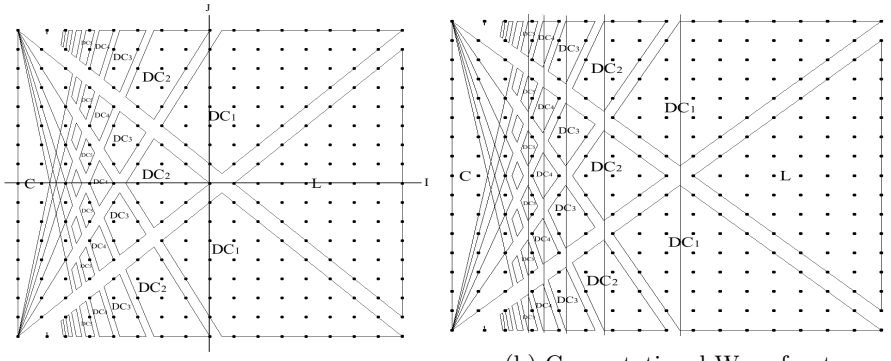
3.2 Case 2: One Dimensional Non-Uniform Reuse Pattern

```
for (i = –N; i <= N; i++)
        for (j = –N; j<= N; j++)
                X[i,j]= Y[i,j] + Y[i,i+j+N];
```

Fig. 6: Loop-Nest with One Dimensional Non-Uniform Reuse

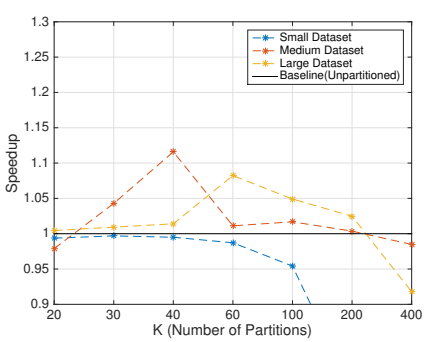
The references to array Y for this case, shown in Fig. 6, are: $\Gamma_{i,\,j} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $\Gamma_{i,\,i+j+N} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \mathbf{I} + \begin{pmatrix} 0 \\ N \end{pmatrix}$. Therefore, the temporal reuse relation $\mathcal{T} = (T, t)$, assumes the following form, according to Equation 2: $T = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$, $t = \begin{pmatrix} 0 \\ -N \end{pmatrix}$. For this case, the maximum value of k is too high. It is dependent on the variable N, which is a representation of the dataset size, as such: $T^k = \begin{pmatrix} 1 & 0 \\ -k & 1 \end{pmatrix}$, $t = \begin{pmatrix} 0 \\ -kN \end{pmatrix}$.

Since the reuse is along the dimension J, refer to Figure 7a, the maximum value that k can reach is $2N - 1$. As the algorithm moves towards $-I$ direction, it forms smaller partitions. This leads to the drawback of the original algorithm. As described in the Section 2.3, an optimal value for k must be chosen such

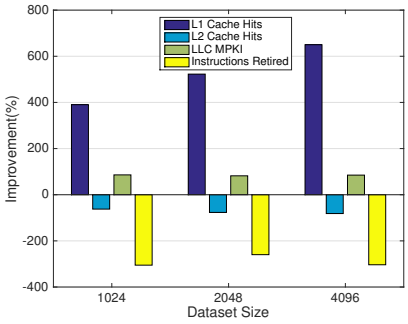


(a) Reducing partition size (b) Computational Wave-front

Fig. 7: Partitions of the iteration space in Case 2.



(a) Speedup vs Number of Partitions.



(b) % of Improvement in L1 and L2 hits, LLC misses and Instructions Retired

Fig. 8: Case 2: Optimal Number of Partitions and Improvement in Counters

that the achievable speedup is not diminished by the excessive control statement overhead. Therefore, the algorithm must halt as soon as tile size reduces below 25×25 iterations. This is deduced from the experimental data as shown in Figure 8a. The optimal value of k was found to be around 30 in a small dataset (N=1024), 40 in a medium dataset (N=2048), and 60 in a large dataset (N=4096). If a value of k is chosen to be lower than the optimal value, the loop execution experiences a performance degradation due to low locality exploitation. On the other hand, if a value of k is chosen to be larger than the optimal value, the loop execution experiences a performance degradation due to control statement overhead.

Another important contribution to the achieved speedup comes from a code generation optimization which is discussed in Section 2.5. If partitions are executed similarly as in Case 1, the control statement overhead will inhibit achieve the maximum speedup achievable. By further splitting and executing them in a variable step wave-front (Fig. 7b) the control overhead is reduced - because the loop boundary conditions are simplified. This method does not conform to the originally proposed method of computing partitions of similar reuse pattern together inside single loop nest. This wave-front method execute different partition types together inside the outer-most loop. It also improves spatial locality due to reuse on same cache-line for multiple partition-types. The increase in cache hits as shown in Fig. 8b is evident of improvement in locality.

On serial execution, the maximum speedup of 1.13x is achieved for the medium dataset (Fig. 11a). Whereas, on parallel execution the speedup improves with the size of the dataset reaching maximum of 2.27x (Fig. 11b).

3.3 Case 3 (Seidel-2D) and Case 4 (Jacobi-2D): Uniform Reuse Pattern and Multiple References

Case 3 and 4 are stencil benchmarks taken from the PolyBench. Case 3 (Seidel stencil) from Fig. 9a has multiple references in 8 directions. Therefore, the heuristics mentioned in Section 2.4 must be applied to choose the best two references for creating partitions. The reuse vectors $\boldsymbol{v}_{i,j-1}$ and $\boldsymbol{v}_{i,j+1}$ link them-

Partial loop−nests exposing reuse

```
for (i = 1; i < N; i++) {
  for (j = 1; j< N; j++) {
  A[i][j]=(A[i-1][j-1]+A[i-1][j]+
           A[i-1][j+1]+A[i][j-1]+
           A[i][j]+A[i][j+1]+
           A[i+1][j-1]+A[i+1][j]+
           A[i+1][j+1])/9.0;
  }
}
```

(a) Seidel-2D

```
for (i = 1; i < N; i++) {
  for (j = 1; j< N; j++) {
    B[i][j]=(A[i][j]+
             A[i][j-1]+
             A[i][1+j]+
             A[1+i][j]+
             A[i-1][j])*0.2;
  }
}
```

(b) Jacobi-2D

Fig. 9: Loop-Nest with Uniform Reuse Pattern and Multiple References

selves by $\mathcal{T}^2$. Therefore, one of the reuse relations must be eliminated. The same applies to $(\boldsymbol{v}_{i-1,j-1},\ \boldsymbol{v}_{i+1,j+1})$,$(\boldsymbol{v}_{i-1,j},\ \boldsymbol{v}_{i+1,j})$ and $(\boldsymbol{v}_{i+1,j-1},\ \boldsymbol{v}_{i-1,j+1})$. Reference $\boldsymbol{v}_{i,j}$ must also be removed since its combination with any other $\boldsymbol{v}$ generates multiple equivalent partitions along $\boldsymbol{v}$. Therefore, references $\boldsymbol{v}_{i,j+1}$, $\boldsymbol{v}_{i+1,j+1}$, $\boldsymbol{v}_{i+1,j}$, $\boldsymbol{v}_{i+1,j-1}$ and $\boldsymbol{v}_{i,j}$ must be eliminated. Also, some pairs of reuse vectors produces partitions which consume the entire iteration space, i.e., the two references $(\boldsymbol{v}_{i-1,j-1},\ \boldsymbol{v}_{i,j-1})$ link every iteration in the domain. Therefore, this pair of references must be eliminated in addition to the pairs $(\boldsymbol{v}_{i-1,j},\ \boldsymbol{v}_{i-1,j-1})$, $(\boldsymbol{v}_{i-1,j},\ \boldsymbol{v}_{i-1,j+1})$ and $(\boldsymbol{v}_{i-1,j-1},\ \boldsymbol{v}_{i-1,j+1})$. Finally, $\boldsymbol{v}_{i,j-1}$ and $\boldsymbol{v}_{i-1,j}$ are left and they create the partitioning as shown in Figure 10.

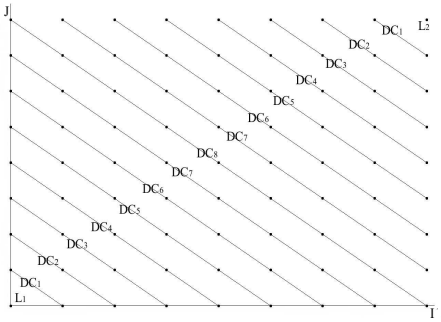


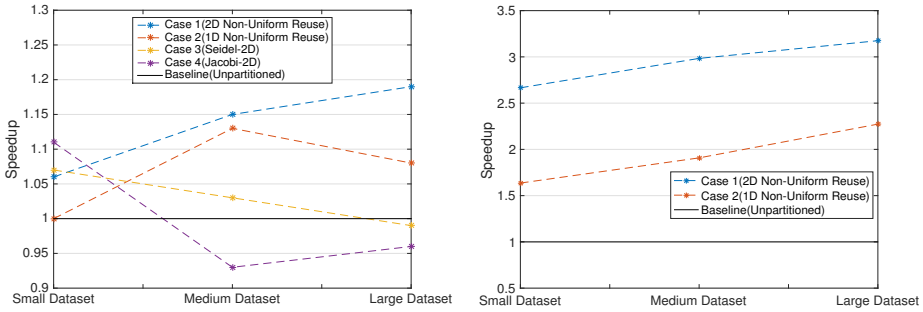
Fig. 10: Partitions for Stencils

The two stencils show different performance results due to different amount of reuse among iterations in the partitions. In the case of Seidel-2D, there is more reuse between consecutive iterations inside a single sub-partition than Jacobi-2D, due to additional reuse on $\boldsymbol{v}_{i+1,j-1}$ and $\boldsymbol{v}_{i-1,j+1}$ in Seidel-2D.

3.4 Improvement in Performance

Serial Execution. The performance of the polygonally tiled code, compiled with LLVM (flags: -O3 -fno-inline-functions), is compared against Polly - an optimizer for LLVM - optimized code (flags: -O3 -polly -polly-vectorizer=stripmine -fno-inline-functions, tile size = 32×32). The lack of benchmarks exhibiting non-uniform reuse pattern in standard benchmarks suites like SPEC CPU and Polybench restricts the comparison of our technique to the existing techniques.

Parallel Execution. The polygonally tiled code is compared for performance against the code optimized using PLuTo-0.11.4 (flags: --tile --parallel --diamond-tile, tile size = 32×32) that generate OpenMP® code. PLuTo is chosen for par-



(a) Speedup (Serial Execution) (b) Speedup (Parallel Execution)

Fig. 11: Performance Improvement

allel execution because it generates better schedules for regular tiles on parallel execution and supports diamond tiling. Both codes are compiled with Intel's ICC-15.0.4 compiler (flags: -O3 -xHost -ansi-alias -ipo -fp-model precise -fno-inline-functions) and are executed across 8 threads.

4 Related work

Loop tiling, its variants and combination with other loop transformations [7, 12, 13] aim to optimize data locality along with other objectives, e.g., exhibiting loop level parallelism [1, 14]. Tiling techniques are concentrated on partitioning the iteration space into group of iterations (tiles) of similar shape and size. The factors determining the size of tiles may depend on memory hierarchy, cache capacities, etc. When execution proceeds tile by tile, reuse distances are no longer a function of the problem size, but a function of the tile size.

Optimal Tile Size and Parametrized Tiling. Determining the tile size at compile-time usually produce suboptimal solution since the cache sizes for the target architecture are not known in many situations. Parameterized tiling techniques [11, 8] have shown that it is possible to get comparable performance and parallelism [6] as compared to statically compile-time generated tiled loop-nests. However, tiling the loops with non-uniform reuse pattern is still a challenge due to the inability of defining a single set of dependency vectors which can govern a tiling size and pattern. Whereas, in our technique the size of the tiles is solely determined by the reuse pattern of the loop-nest.

Modern Tiling Geometries. In addition to the variable sized tiles, some recent work on the exploration of newer tiling geometries have shown some promise, especially for stencil computations. The work in [2] shows that diamond-shaped tiles - when executed in parallel - can achieve concurrent start for the tiles which might not have been possible with regular rectangular/parallelogram tiles. Tiling in the shape of variable-sized Hexagons [5] provides better locality and concurrent execution of tiles for parallel architectures like GPUs. But, varying tile shapes for better locality has not received similar attention. The polygonal tiling technique presented in this work is not bound to a specific tile shape. Instead, tile shapes are determined based on the iteration space's reuse pattern.

5 Conclusion

In this work, a polygonal tiling technique is presented, which is not constrained to either the shape or the size of tiles that needs to be pre-determined. The shapes and sizes are governed by the reuse pattern of the loop-nests. The proposed technique partitions the iteration space and schedule the partitions to maximize locality.

Our experiments on a set of loops exhibiting either non-uniform or uniform reuse patterns show that a significant portion of the achievable speedup is missed when applying traditional loop tiling to such loops. Speedup is significant for loops with non-uniform reuse pattern on serial execution as shown in the case studies. Benefits of the presented polygonal tiles is even greater for multi-threaded execution for such loops. High speedup (up to 3.17x) is achieved and it consistently improves on increasing the input size.

References

1. Agarwal, A., et al.: Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. TPDS '95 6(9), 943–962 (Sep 1995)
2. Bandishti, V., et al.: Tiling Stencil Computations to Maximize Parallelism. In: SC '12. pp. 40:1–40:11. IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
3. Bastoul, C.: Code Generation in the Polyhedral Model Is Easier Than You Think. In: PACT '13. pp. 7–16. Juan-les-Pins, France (September 2004)
4. Bondhugula, U., et al.: A Practical Automatic Polyhedral Program Optimization System. In: PLDI (Jun 2008)
5. Grosser, T., et al.: Hybrid Hexagonal/Classical Tiling for GPUs. In: CGO '14. pp. 66:66–66:75. ACM, New York, NY, USA (2014)
6. Hartono, A., et al.: DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In: IPDPS '10. pp. 1–12 (April 2010)
7. Irigoin, F., Triolet, R.: Supernode Partitioning. In: POPL '88. pp. 319–329. ACM, New York, NY, USA (1988)
8. Kim, D., et al.: Multi-level tiling: M for the price of one. In: SC '07. pp. 1–12 (Nov 2007)
9. Meister, B., Loechner, V., Clauss, P.: The Polytope Model for Optimizing Cache Locality. Tech. rep., Technical Report RR 00-03, ICPS-LSIIT (2000)
10. Padua, D.A., Wolfe, M.: Advanced Compiler Optimizations for Supercomputers. Commun. ACM 29(12), 1184–1201 (1986)
11. Renganarayanan, L., et al.: Parameterized Tiled Loops for Free. In: PLDI '07. pp. 405–414. ACM, New York, NY, USA (2007)
12. Wolfe, M.: Iteration Space Tiling for Memory Hierarchies. In: Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing. pp. 357–361. SIAM, Philadelphia, PA, USA (1989)
13. Wolfe, M.: More Iteration Space Tiling. In: SC '89. pp. 655–664. ACM, New York, NY, USA (1989)
14. Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers, Norwell, MA, USA (2000)

Acknowledgments. We would like to thank Benoît Meister and Vincent Loechner for providing us with their implementation which laid the foundation for this work. This work was supported in part by NSF award XPS 1533926.