Automatic Intra-Array Storage Optimization

SOMASHEKARACHARYA G. BHASKARACHARYA, Indian Institute of Science
UDAY BONDHUGULA, Indian Institute of Science
ALBERT COHEN, INRIA

IISc-CSA-TR-2014-3
August 2014 (Revised: November 2014)

Efficient memory allocation is crucial for data-intensive applications as a smaller memory footprint ensures better cache performance and allows one to run a larger problem size given a fixed amount of main memory. In this paper, we describe a new automatic storage optimization technique to minimize the dimensionality and storage requirements of arrays used in sequences of loop nests with a pre-determined schedule. We formulate the problem of intra-array storage optimization as one of finding the right storage partitioning hyperplanes: each storage partition corresponds to a single storage location. Our heuristic is driven by a dual objective function that minimizes both, the dimensionality of the mapping and the extents along those dimensions. The technique is dimension optimal for most codes encountered in practice. The storage requirements of the mappings obtained also are asymptotically better than those obtained by any existing schedule-dependent technique. Storage reduction factors and other results we report from an implementation of our technique demonstrate its effectiveness on several real-world examples drawn from the domains of image processing, stencil computations, high-performance computing, and the class of tiled codes in general.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, optimization*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Compilers, storage mapping optimization, memory optimization, array contraction, polyhedral framework

1. INTRODUCTION AND MOTIVATION

Efficient storage management for array variables in a program requires that memory locations be reused as much as possible, thereby minimizing their storage requirement. Consider a statement, which writes to an array, appearing within an arbitrarily nested loop. Two dynamic instances of the statement can store values that they compute to the same memory location provided the lifetimes of these values do not overlap. Therefore, most solutions to this problem are schedule-dependent. Storage optimization can be performed soon after execution reordering transformations have been applied, but before generating the final transformed code.

Automatic storage optimization is crucial for data-intensive applications. In several cases, a programmer is particularly interested in running a dataset while utilizing the entire main memory capacity of a system. In such cases, performance (execution time) is secondary. Storage optimization thus allows a programmer to run a larger problem size for a given main memory capacity. When using multiple applications, it also allows more applications to fit in memory. In addition, storage optimization can also potentially improve performance as a direct result of a smaller memory footprint. Storage optimization has also proved to be a critical optimization for domain-specific compilers. Image processing pipelines [Ragan-Kelley et al. 2013] and stencil computations are two example domains where code generators rely on analysis to reduce the peak memory usage of the generated code. Compilers for functional languages with arrays, or dataflow languages with single-assignment semantics also need copy-avoidance to maximize memory reuse [Abu-Mahmeed et al. 2009].

The scope of programs that we consider for this work is a class of codes known as *affine loop nests*. Affine loop nests are sequences of arbitrarily nested loops (perfect or imperfect) where data accesses and loop bounds are affine functions of loop iterators and program parameters (symbols which do not vary within the loop nest). Due to the affine nature of data accesses, these loop program portions are statically predictable and can be analyzed and transformed using the polyhedral compiler framework [Aho et al. 2006]. Significant advances have been made in memory optimization for affine loop nests or its restricted forms [Lefebvre and Feautrier 1998; Strout et al. 1998; Thies et al. 2001; Darte et al. 2005; Alias et al. 2007]. However, we first show that a good memory optimization technique is still missing. The solutions found by existing works for several commonly encountered cases are far from good or optimal and could even miss nearly all storage optimization potential.

The storage optimization problem for arrays can be viewed as contracting the array along one or more dimensions to fixed sizes, or contracting along directions different from the *directions* along which the array is originally indexed. Thus, an approach to contraction can be viewed as one that finds: (1) good directions along which to contract (and the order in which to contract) in case the original ones are not good, and (2) the minimal sizes to which each of the chosen dimensions can be contracted. While the latter part was first comprehensively studied by [Lefebvre and Feautrier 1998], there is no heuristic available to obtain good solutions to the former. Choosing the right directions and their ordering impacts both the dimensionality of the resulting storage and the storage size. For example, it can be the difference between say N^2, $2N$, and N storage for what was originally an NxN array. [Lefebvre and Feautrier 1998; Darte et al. 2005] had either worked with the canonical (original) basis or assumed that the right directions would be provided by an oracle.

Our scheme computes storage allocations suitable for a pre-determined multi-dimensional schedule for the iterations. Typically, one determines a schedule based on criteria like locality, parallelism and potentially, even memory footprint. It is thus natural and reasonable to assume that the schedule has been fixed by the time storage contraction is ready to be performed. Our approach finds directions that minimize the dimensionality of the contracted storage We introduce the notion of a *storage partitioning hyperplane*: such hyperplanes define a partitioning of the iteration space such that each partition uses a single memory location. Our approach is then of iteratively finding a minimum number of *storage partitioning hyperplanes* with certain criteria. The objectives ensure the right orientation of the storage hyperplanes such that the dimensionality of the contracted array is as low as possible, and for each of those dimensions, its extent is minimized.

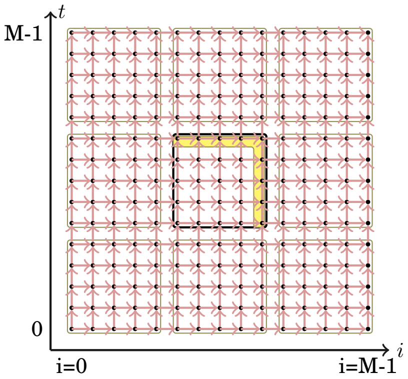


Fig. 1.

Consider the stencil computation with dependences $(1, 0)$, $(0, 1)$ in Fig. 1. It corresponds to the tiled version of the code in Fig. 2. For a given tile, only its top and right boundaries are live-out. As the primary objective behind tiling for locality is to exploit reuse in multiple directions while the data accessed fits in faster memory, live-out sets along two or more boundaries are common with tiling. In Fig. 1, for a schedule that iterates row-wise within a tile, indexing the array along the canonical directions does not reduce storage, i.e., if T is the tile size,

T^2 storage per tile is needed. This solution corresponds to the canonical storage hyperplanes $(1, 0)$ and $(0, 1)$. The contraction factors obtained by [Lefebvre and Feautrier 1998] would just be N along each of the two dimensions. None of the heuristics described in [Darte et al. 2005; Alias et al. 2007] find a different basis. If the array is partitioned along the hyperplane $(1, -1)$, i.e., if all points (t, i) in the array such that $t - i = constant$ reuse the same memory location, the tile can be executed using a storage of just $2T - 1$ cells. The storage buffer would finally hold the $2T - 1$ live-out values. An access $A[i, j]$ will be transformed to an access $A[(t - i) \bmod (2T - 1)]$, and this is also the optimal solution. The occupancy vector based approach of [Strout et al. 1998] does obtain this optimal storage, but it is designed for perfect loop nests with constant dependences, and its scheduled independent nature leads to sub-optimal solutions in general. The schedule-dependent approach we develop in this paper finds the optimal storage mapping in this case automatically, and works for general affine loop nests. Other dependence patterns or more complex tiling shapes can lead to non-trivial mappings which are very hard to derive by hand.

In summary, our contributions are the following:

- We describe a new technique for storage optimization while casting the latter as an array space partitioning problem, where each partition uses the same memory location. We then formulate an ILP problem solvable using a greedy heuristic whose objective takes into account both – the dimensionality of the mapping and the extent along each dimension.
- We implement and evaluate our technique on various domain-specific benchmarks and demonstrate reductions in storage requirement ranging from a constant factor to asymptotic in the extents of the original array dimensions or loop blocking factors.
- Owing to the similarity to the nature of our approach, a link is established between storage optimization and loop transformation algorithms in the polyhedral model – both of them can be seen as partitioning problems. There were advances in the latter over the past two decades while no comparably strong heuristics existed for memory optimization.

Section 2 provides the necessary background on the framework used, the successive modulo technique of [Lefebvre and Feautrier 1998] and introduces the notation used later. Section 3 details our storage optimization scheme. Various examples from real-world applications are discussed in Section 4. Section 5 reports results from our implementation. Related work and conclusions are presented in Section 6 and Section 7 respectively.

2. BACKGROUND

This section provides the background and notation for the techniques we present for storage mapping optimization.

Definition 2.1. The set of all vectors $\vec{v} \in \mathbb{Z}^n$ such that $\vec{h}.\vec{v} = k$ constitute an affine hyperplane.

Different constant values for k generate different parallel instances of the hyperplane which is usually characterized by the vector, $\vec{h}$, normal to it.

Polyhedral representation. The polyhedral representation of a program part is a high-level mathematical representation convenient for reasoning about loop transformations. The class of programs that are typically represented in this model are affine loop nests. Each execution instance $\vec{i_S}$ of a statement S, within n enclosing loops, is represented as an integer point in an n-dimensional polyhedron, which defines the *iteration domain* D of the statement. A multi-dimensional affine scheduling function θ

maps each point in the iteration domain to a multi-dimensional time point. Read and write accesses to an array variable with an m-dimensional array space are represented by affine array access functions which map the iteration space of the statement to the array's data space.

Farkas' Lemma. Several polyhedral techniques rely on the application of the affine form of the Farkas' lemma [Schrijver 1986; Feautrier 1992].

LEMMA 2.2. *Let D be a non-empty polyhedron defined by s affine inequalities or faces: $\vec{a}_k.\vec{x} + b_k \geq 0$, $1 \leq k \leq s$. An affine form $\psi(\vec{x})$ is non-negative everywhere in D iff it is a non-negative combination of the faces, i.e.,*

$$\psi(\vec{x}) = \lambda_0 + \Sigma_{(k=1)}^{(k=s)} \lambda_k(\vec{a_k}.\vec{x} + b_k), \lambda_k \geq 0. \tag{1}$$

The λ_ks are known as Farkas multipliers.

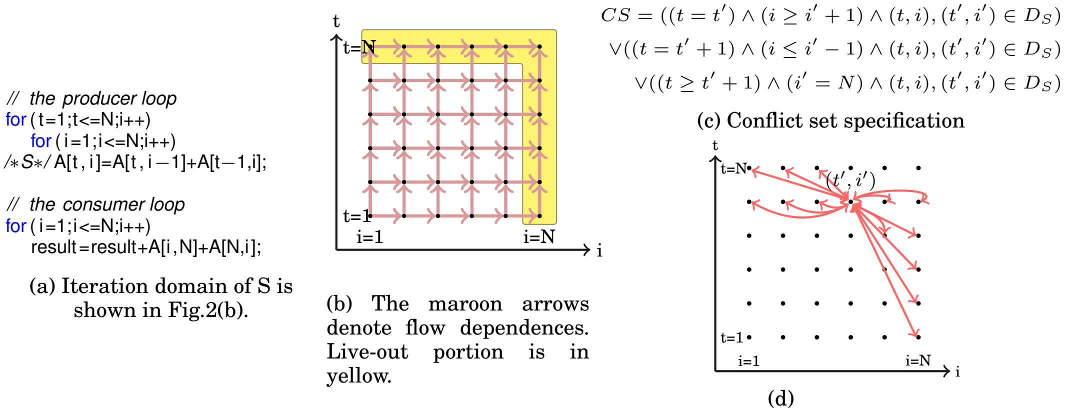
2.1. Successive Modulo Technique

[Lefebvre and Feautrier 1998] proposed a storage optimization technique which they referred to as partial data expansion. A given static control program is subjected to array dataflow analysis and then converted into functionally equivalent single-assignment code so that all the artificial dependences (output and anti) are eliminated. The translation to single-assignment code involves rewriting the program so that each statement S writes to its own distinct array space A_S, which has the same size and shape as that of the iteration domain of S. Without any loss of generality, if we assume that the loop indices are non-negative, then $\vec{i}_S$ writes to $A_S[\vec{i}]$. This process of expanding the array space is known as *total data expansion*. A schedule θ is then determined for the single-assignment code.

In order to alleviate the considerable memory overhead incurred due to such total expansion, the array space is then contracted along the axes represented by the loop iterators. This *partial expansion* technique is based on the notion of the *utility span* of a value computed by a statement instance $\vec{i}_S$ at time $\theta(\vec{i}_S)$ to a memory cell C. It is defined to be the sub-segment of the schedule during which the memory cell C is active, i.e., the value stored at C still has a pending use. Suppose that the last pending use of the value in C occurs in iteration $L(\vec{i}_S)$, at logical time $\theta(L(\vec{i}_S))$. Any new output dependence which does not conflict with the flow dependence between $\vec{i}_S$ and $L(\vec{i}_S)$ corresponding to the time interval $[\theta(\vec{i}_S), \theta(L(\vec{i}_S))]$, is an output dependence that can be safely introduced.

Definition 2.3. Two array indices $\vec{i}, \vec{j}$ such that $\vec{i} \neq \vec{j}$ conflict with each other and the conflict relation $\vec{i} \bowtie \vec{j}$ is said to hold iff $\theta(\vec{i}_S) \preceq \theta(L(\vec{j}_S))$ and $\theta(\vec{j}_S) \preceq \theta(L(\vec{i}_S))$ are both true, i.e., if the corresponding array elements are simultaneously live under the given schedule θ.

The conflict set CS is the set of all pairs of conflicting indices given by $CS = \{(\vec{i}, \vec{j}) \mid \vec{i} \bowtie \vec{j}\}$. In accordance with the above definition, the conflict relation $\bowtie$ is a symmetric, non-reflexive. Partial expansion is performed iteratively with each statement being considered once at every depth of the surrounding loop-nest. The *contraction modulo* e_p (or expansion degree as Lefebvre et al refer to it), along the axis of the array space which corresponds to the loop at depth p, is computed as follows. Suppose DS is the set of differences of indices which conflict, i.e., $DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$. Similarly, let $DS_p = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j} \wedge \vec{i} \succ \vec{j} \wedge (i_x = j_x \forall x < p)\}$. If $\vec{b}$ is the lexicographic maximum of DS_p, the contraction modulo is given by $e_p = b_p + 1$, where $b_p\hat{u}_p$ is the component of $\vec{b}$



```
// the producer loop
for (t=1;t<=N;i++)
    for (i=1;i<=N;i++)
/*S*/ A[t,i]=A[t,i-1]+A[t-1,i];

// the consumer loop
for (i=1;i<=N;i++)
    result=result+A[i,N]+A[N,i];
```

(a) Iteration domain of S is shown in Fig.2(b).

(b) The maroon arrows denote flow dependences. Live-out portion is in yellow.

$$
\begin{aligned}
CS = &((t = t') \wedge (i \geq i' + 1) \wedge (t, i), (t', i') \in D_S)\\
&\vee((t = t' + 1) \wedge (i \leq i' - 1) \wedge (t, i), (t', i') \in D_S)\\
&\vee((t \geq t' + 1) \wedge (i' = N) \wedge (t, i), (t', i') \in D_S)
\end{aligned}
$$

(c) Conflict set specification

(d)

Fig. 2. The geometrical representation in Fig.2(d) shows the array space A written to by statement S in the code snippet shown in Fig.2(a). The red double-headed arrows in Fig.2(d) denote the various conflicts associated with the array index (t', i').

along the axis i_p, with $\hat{u}_p$ representing the unit vector along the same axis. In essence, the contraction modulo e_p represents the degree of contraction along that axis. The final storage mapping is obtained by converting it into a modulo mapping so that the statement instance $\vec{i}_S$ writes to $A_S[\vec{i} \bmod \vec{e}]$, where $\vec{e} = (e_0, e_1, \ldots, e_{n-1})$. This method to determine the contraction moduli will hereafter be referred to as the *successive modulo technique*.

3. INTRA-ARRAY STORAGE OPTIMIZATION

In this section, we present all details of our storage optimization technique.

3.1. A simple example

The successive modulo technique is quite versatile, scalable and also parametric. However, the eventual modulo storage mapping obtained does not always lead to minimal storage requirements. Consider the static control loop-nests in Fig. 2. The producer loop-nest is already in single-assignment form where each statement instance $S(t, i)$ writes to its own distinct memory cell $A[t, i]$ so that the array space A has the same size and shape as the iteration domain of statement S. Suppose the schedule determined is $\theta(t, i) = (t, i)$. There are some values computed by statement S which are live even after all its instances have been executed. These live-out values reside in the set of memory cells, $\{(t, i) \mid (t, i) \in A \wedge (i = N) \vee (t = N)\}$. As a result, the conflict set CS is made up of conflicts not only due to the uniform lifetimes of the non-live-out values but also due to the non-uniform lifetimes of the live-out values. Specifically, the array index associated with a live-out value conflicts with the array index associated with any value computed later in the schedule. So, the resulting conflict set CS, of pairs of conflicting indices $(t, i) \bowtie (t', i')$, is a union of convex polyhedra characterized by the constraints shown in Fig.2(c).

The conflict relation is, strictly speaking, symmetric. For brevity, the constraints in Fig.2(c) represent a conflict between a pair of conflicting indices only once, effectively treating it as an unordered pair. The first two disjuncts in Fig.2(c) together represent conflicts due to the flow dependence $(1, 0)$, which is also the maximum utility span of any non-live-out value (Fig.2(b) and Fig.2(d)). The last disjunct expresses the conflicts due to the live-out values. Often, the last read of the value computed by one statement instance and the write by another instance of the same statement occur at the same logical time. Hereafter, in such scenarios, we do not treat the associated indices as conflicting since they can be mapped to the same memory cell, e.g. in Fig.2(c), (t', i') and $(t' + 1, i')$ do not conflict.

Applying the successive modulo technique, at loop-depth $p = 0$, the contraction modulo obtained is $e_0 = N$ due to the conflict $(N, N) \bowtie (1, N)$. Similarly, the contraction modulo at loop-depth $p = 1$ is $e_1 = N$ due to the conflict $(N, N) \bowtie (N, 1)$. The resulting modulo storage mapping of $A[t, i] \to A[t \bmod N, i \bmod N]$ requires N^2 storage.

A careful analysis reveals that a better storage mapping for the above example would be $A[t, i] \to A[(i - t) \bmod (2N - 1)]$. This mapping not only ensures that all the intermediate values computed are available until their last uses but also that the live-out values are available even after the producer loop has terminated. Furthermore, it drastically reduces the storage requirement from $O(N^2)$ to $O(N)$, requiring just a single row of $2N - 1$ cells. The above example shows that a straightforward computation of the contraction moduli along the canonical bases can lead to a solution which can be considerably worse than the optimal solution. As will be explained in the following sections, a better approach is to find hyperplanes which partition the array space based on the conflict set and to then use the hyperplane normals as the bases for computing the contraction moduli.

3.2. Storage Hyperplanes and Conflict Satisfaction

We formalize here the notion of a storage partitioning hyperplane (or storage hyperplane) satisfying a conflict $\vec{i} \bowtie \vec{j}$ in the conflict set CS.

Definition 3.1. A conflict between a pair of array indices $\vec{i}$ and $\vec{j}$ is said to be satisfied by a hyperplane $\vec{\Gamma}$ iff $\vec{\Gamma}.\vec{i} - \vec{\Gamma}.\vec{j} \neq 0$.

Essentially, if the hyperplane is thought of as partitioning the array space, a conflict is only satisfied if the array indices involved are mapped to different partitions.

The successive modulo technique can also be understood through this notion of conflict satisfaction. Consider again loop-nest in Fig.2. As explained earlier, the contraction modulo $e_0 = N$ is due to the conflict $(N, N) \bowtie (1, N)$. This is equivalent to the hyperplane $(1, 0)$ partitioning the array space into N partitions. Clearly, the conflicting indices $(1, N)$ and (N, N) are mapped to different partitions, thus satisfying the conflict. The hyperplane $(1, 0)$ satisfies all the conflicts represented by the second and third disjuncts in Fig.2(c). The conflicts specified by the first disjunct are not satisfied as the conflicting indices get mapped to the same partition. However, these conflicts are satisfied at loop-depth $p = 1$. This can be seen as the hyperplane $(0, 1)$ further partitioning each of the N partitions obtained earlier into N distinct sub-partitions. As a result, the conflicting indices in the conflicts which were not satisfied at the previous level get mapped to different partitions. In essence, the successive modulo approach can also be understood as conflict satisfaction being performed by successively partitioning the array space using a series of storage hyperplanes.

The dimensionality of the array space is a loose upper bound on the number of storage hyperplanes required to satisfy all the conflicts. This is because, in the trivial case, the hyperplanes could simply correspond to those which have the canonical axes as their normals. In fact, this is exactly how the modulo storage mapping is determined in the successive modulo technique. In Fig.2, all the conflicts were satisfied using the two canonical hyperplanes, $(1, 0)$, $(0, 1)$. However, the mapping $A[t, i] \to A[(i - t) \bmod (2N - 1)]$, which is better than the resulting solution not only in terms of the storage size required but also in terms of its dimensionality, shows that it is possible to satisfy all the conflicts in the conflict set (Fig.2(c)) using just one storage hyperplane. Generally, the choice of partitioning hyperplanes affects both the dimensionality as well as the storage requirements of the resulting modulo storage mapping.

3.3. A Partitioning Approach

The problem of intra-array storage optimization for a given statement S with an n-dimensional iteration domain D, writing to an array space A (of the same size and shape as D due to total data expansion), can be seen as a problem of finding a set of m partitioning hyperplanes $\vec{\Gamma}_1, \vec{\Gamma}_2, \ldots, \vec{\Gamma}_m$, which together satisfy all conflicts in the conflict set CS i.e., every conflict must be satisfied by at least one of the m hyperplanes. The resulting m-dimensional modulo storage mapping would be of the form $A[\vec{i}] \rightarrow A[M\vec{i} \bmod \vec{e}]$ where M is the $m \times n$ transformation matrix constructed using the m storage hyperplanes as the m rows of the matrix. If a hyperplane $\Gamma_i = (\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,n})$, then the storage mapping matrix M is an $m \times n$ matrix with the i^{th} row $(\gamma_{i,1}\ \gamma_{i,2}\ \cdots\ \gamma_{i,n})$.

$$M = \begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdot & \cdot & \gamma_{1,n} \\ \gamma_{2,1} & \gamma_{2,2} & \cdot & \cdot & \gamma_{2,n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \gamma_{m,1} & \gamma_{m,2} & \cdot & \cdot & \gamma_{m,n} \end{pmatrix}$$

The contraction moduli computed along the normals of the m hyperplanes form the m components of the vector $\vec{e}$.

3.3.1. Conflict Set Specification. The conflict set can be specified as a union of convex polyhedra, also called *conflict polyhedra*, e.g., the disjunction in Fig.2(c). Each integer point in a conflict polyhedron represents a particular conflict. The symmetricity of the conflict relation can be used to simplify the conflict set significantly. Consider a 1-d array space A where all array indices conflict with all other indices. The conflict set CS is then the set of ordered pairs (i, i') such that $i \bowtie i'$ holds. A conflict relation $1 \bowtie 2$ can be encoded as the integer point $(1, 2)$ in the conflict polyhedron $\{i < i' \mid i, i' \in A\}$. Strictly speaking though, if all conflict relations are to be represented, due to the symmetry, another conflict polyhedron $\{i > i' \mid i, i' \in A\}$ would be required to accommodate the conflict relation $2 \bowtie 1$. However, satisfying the conflict $1 \bowtie 2$ implies that $2 \bowtie 1$ is also satisfied as both of them represent the same pair of indices. The second conflict polyhedron is, in effect, redundant in the conflict set. Hereafter, we follow the convention that if a conflict relation $\vec{i} \bowtie \vec{j}$ is represented in a conflict set CS, then CS does not contain a redundant representation of the relation $\vec{j} \bowtie \vec{i}$ as well. There may be multiple ways to specify a conflict set as a union of conflict polyhedra. We follow the convention that the conflict set specification is minimal in the sense that no two conflict polyhedra exist in the union such that their union is itself convex.

3.4. Finding a Storage Hyperplane

In the scenario when the conflict set is empty to begin with, the optimal allocation is to contract the array down to a single scalar variable. Storage hyperplanes only need to be found when the conflict set is non-empty. Suppose there are l conflict polyhedra $K_1, K_2, \ldots, K_l$ so that the conflict set $CS = \cup_{i=1}^{i=l} K_i$. Consider a pair of conflicting indices $\vec{s}$ and $\vec{t}$. By Definition 3.1, a hyperplane $\vec{\Gamma}$ satisfies this conflict if $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \neq 0$. This can be expressed by the disjunction:

$$(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1 \quad \vee \quad (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1.$$

Furthermore, since the iteration space D (and consequently, the array space A) is bounded, there must exist a finite upper bound of the form $(\vec{u}.\vec{P}+w)$ on $(\vec{\Gamma}.\vec{s}-\vec{\Gamma}.\vec{t})$, where

$\vec{P}$ is the vector of program parameters. Such a bound has been used in [Feautrier 1992] and in [Bondhugula et al. 2008], although in different contexts. Additionally, as the conflict relation is symmetric, the upper bound is applicable to the absolute value of the conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$. So, the bounding constraints can be expressed as follows:

$$-(\vec{u}.\vec{P} + w) \ \leq \ (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \ \leq \ (\vec{u}.\vec{P} + w). \tag{2}$$

3.4.1. Encoding Satisfaction with Decision Variables. A storage hyperplane $\vec{\Gamma}$ may not necessarily satisfy all conflicts in the conflict set CS. It may not even satisfy all the conflicts represented in a particular conflict polyhedron. So, in general, $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$ could be positive, negative, or equal to zero. This nature of conflict satisfaction can be captured adequately by introducing a pair of binary decision variables x_{1i}, x_{2i} for each conflict polyhedron K_i such that:

$$x_{1i} \ = \ \begin{cases} 1 & \text{if } (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1, \ \forall \, \vec{s} \bowtie \vec{t} \ \in \ K_i, \\ 0 & \text{if otherwise.} \end{cases}$$

$$x_{2i} \ = \ \begin{cases} 1 & \text{if } (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1, \ \forall \, \vec{s} \bowtie \vec{t} \ \in \ K_i, \\ 0 & \text{if otherwise.} \end{cases}$$

Note that the binary decision variables x_{1i}, x_{2i} indicate the nature of conflict satisfaction at the granularity level of a conflict polyhedron and not at the granularity level of each conflict. Even if there exists one conflict in the conflict polyhedron which is not satisfied by the hyperplane, then the conflict polyhedron, as a whole, is still treated as unsatisfied. So, the constraint that $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$ could be positive, negative, or equal to zero can be expressed as the conjunction:

$$\begin{aligned} & (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1 - (1 - x_{1i})(\vec{u}.\vec{P} + w + 1) \\ \wedge \quad & (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1 + (1 - x_{2i})(\vec{u}.\vec{P} + w + 1). \end{aligned} \tag{3}$$

By definition, x_{1i} and x_{2i} cannot be simultaneously equal to 1. Such a scenario would mean that the constraints in the above conjunction would contradict each other. However, if $x_{1i} = 1$ and $x_{2i} = 0$, then the first conjunct degenerates into the conflict satisfaction constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1$ whereas the second conjunct is reduced to the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq (\vec{u}.\vec{P} + w)$, which is implied by the bounding constraints (2). Similarly, if $x_{2i} = 1$ and $x_{1i} = 0$, the first conjunct becomes $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq -(\vec{u}.\vec{P} + w)$ which is again implied by the bounding constraint (2). The second conjunct degenerates into the conflict satisfaction constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1$. When there is still at least one conflict which remains unsatisfied, both x_{1i} and x_{2i} must be equal to 0. In such a scenario, it can be seen that neither of the two conjuncts degenerate into a conflict satisfaction constraint. Instead, the entire conjunction boils down to the bounding constraints (2), which must always hold, regardless of whether all or a few of the conflicts in the conflict polyhedron are satisfied.

Each of the l conflict polyhedra is associated with its own pair of binary decision variables, both of which cannot simultaneously be equal to one. So, the number of conflict polyhedra all of whose conflicts are satisfied by a hyperplane can be estimated as the sum of all the decision variables:

$$\eta = \Sigma_{i=1}^{i=l}(x_{1i} + x_{2i}). \tag{4}$$

The number η forms the basis of our greedy heuristic for finding good storage hyperplanes. Greater the value of η, fewer the number of conflict polyhedra whose conflicts still remain unsatisfied. Consequently, it is likely that fewer storage hyperplanes will be needed to satisfy the remaining conflicts. A particularly interesting case is when η can be made to equal l. The storage hyperplane found then would have satisfied all conflicts on its own without the need to find any more hyperplanes. In other words, maximizing η serves as a reasonably good greedy approach for minimizing the number the storage hyperplanes and thereby, the dimensionality of the final storage mapping.

3.4.2. Linearizing the Constraints. The storage hyperplane $\vec{\Gamma}$ should be such that the bounding constraints (2) hold at every integer point $\vec{v}$ in a conflict polyhedron. Each conjunct in the bounding constraints can be rewritten to be in the form $\psi(\vec{v}) \geq 0$ where $\psi(\vec{v})$ is affine. By the Farkas' lemma (1), the affine form can be equated to a non-negative linear combination of the faces of the conflict polyhedron. The loop variables can then be eliminated by equating their respective coefficients to obtain an equivalent set of linear inequalities involving only the coefficients, some of which are the Farkas' multipliers. However, the same procedure cannot be repeated for the decision constraints in (3) as neither of the two conjuncts can be rewritten in the form $\psi(\vec{v}) \geq 0$ (cf (1)). The coefficients of $\vec{P}$ in both conjuncts are products of a decision variable and $\vec{u}$'s coefficients, and similarly $x_{1i}w$ and $x_{2i}w$ are non-linear. Therefore, the decision constraints in (3) cannot be linearized using the Farkas' lemma.

However, since $(\vec{u}.\vec{P} + w)$ is finite, there must exist a finite upper bound on it of the form $(c\vec{P} + c)$, i.e.,

$$\begin{aligned} &(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq (\vec{u}.\vec{P} + w) \leq (c\vec{P} + c) \\ \wedge \quad &-(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq (\vec{u}.\vec{P} + w) \leq (c\vec{P} + c). \end{aligned} \tag{5}$$

In practice, a high value such as $c = 100$ (higher if no parameters exist and all loop bounds are known at compile time) gives a reasonably good estimate of c, allowing c to be treated as a suitably chosen constant value. Each individual constraint in (5) can be treated using Farkas' lemma to obtain a set of equivalent linear inequalities after eliminating the loop variables. Due to transitivity, $(c\vec{P} + c)$ is also an upper bound on $|\,\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\,|$.

The decision constraints in (3) were formulated such that if either x_{1i} or x_{2i} is equal to 1, then one of the conjuncts degenerates into a conflict satisfaction constraint while the other into one of the bounding constraints in (2), which specify $(\vec{u}.\vec{P} + w)$ as an upper bound on $|\,\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\,|$. Now, along similar lines, an alternative set of decision constraints can be formulated as follows:

$$\begin{aligned} &(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1 - (1 - x_{1i})(c\vec{P} + c + 1) \\ \wedge \quad &(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1 + (1 - x_{2i})(c\vec{P} + c + 1). \end{aligned} \tag{6}$$

The key is that c is a constant, and it now makes the conflict satisfaction constraints amenable to linearization through application of Farkas' lemma. Although the variables $\vec{u}$ and w still feature in the expanded set of bounding constraints in (5), the decision constraints in (6) are now devoid of them. Note that $c\vec{P} + c$ has been substituted for $\vec{u}\vec{P} + w$ in (3) alone to obtain (6). The difference between (3) and (6) is only with respect to the upper and lower bound that is imposed on $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$ when one of the binary decision variables x_{1i}, x_{2i} is equal to 1 or when both are equal to 0. It still holds that x_{1i} and x_{2i} cannot simultaneously be equal to 1. The bound $c\vec{P} + c$ only has to be sufficiently large to bound the conflict difference – it need not be tight and it does not influence the objectives we will propose and the solutions obtained in any way.

3.4.3. A Greedy Double-Objective. The resulting ILP system consists of the constraints obtained due to the expanded set of bounding constraints in (5), the revised decision

Algorithm 1 Find a modulo storage mapping given a non-empty conflict set CS for the array space A; $\vec{P}$: the vector of program parameters.

```
1:  procedure FIND-MODULO-MAPPING($A$, $CS$, $\vec{P}$)
2:      $CS' \leftarrow CS$
3:      $m \leftarrow 0$
4:      while $CS' \neq \varnothing$ do
5:          $m \leftarrow m + 1$
6:          $(\Gamma_m, e_m) \leftarrow$ FIND-NEXT-HYPERPLANE($CS'$)
7:          Revise the conflict set ($CS'$) as shown in (9) by revising the conflict poly-
            hedra as shown in (8)
8:      Let $M$ be the transformation matrix constructed with hyperplanes
        $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$ forming its rows
9:      Let $\vec{e}$ be the vector of contraction moduli $e_1, e_2, \ldots, e_m$
        return $(M, \vec{e})$
10: procedure FIND-NEXT-HYPERPLANE($CS'$)
11:     $C \leftarrow \varnothing$
12:     for all conflict polyhedra $K'_i \in CS'$ do
13:         Formulate bounding constraints as shown in (5)
14:         Formulate satisfaction decision constraints as shown in (6)
15:         Apply Farkas' lemma to each of the above constraints (formulated in steps
            13 and 14) to obtain an equivalent set of linear equalities/inequalities
16:         Add the linear inequalities/equalities to $C$
17:     Add the constraint on $\eta$ shown in (4) to $C$
18:     Compute lexicographic minimal solution as shown in (7) to obtain the hyper-
        plane $\Gamma$ and the corresponding contraction modulo $e$
        return $(\Gamma, e)$
```

constraints in (6) and also the constraint on η given by (4). Such constraints are derived for each of the l conflict polyhedra. The greedy approach is to determine a storage hyperplane $\vec{\Gamma}$ such that the estimated number of conflict polyhedra η, all of whose conflicts are satisfied is maximized. This affects the dimensionality of the storage mapping which is eventually obtained.

Another factor which needs to be considered while determining the storage hyperplanes is the storage size of the resulting modulo storage mapping. The storage size of a modulo storage mapping determined using the successive modulo technique is computed as the product of the contraction moduli. In the successive modulo technique, the contraction moduli are computed along the canonical bases. Essentially, the canonical bases also serve as the storage hyperplane normals. In general though, the storage hyperplane normals may not necessarily correspond to the canonical bases. However, the modulo can still be computed based on the maximum conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$, which is essentially the maximum component of any conflict difference $(\vec{s} - \vec{t})$ along the normal of the hyperplane $\vec{\Gamma}$. Since the contraction modulo is 1 plus the maximum conflict difference, greater the maximum conflict difference, more is the storage size required for the resulting storage mapping. Therefore, in order to minimize the storage size, another objective in solving the ILP system is to minimize the maximum conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$. As $(\vec{u}.\vec{P} + w)$ is an upper bound on $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$, this can be done by minimizing $(\vec{u}.\vec{P} + w)$.

The number η is at most equal to l. If $\eta' = (l - \eta)$, the double-objective of maximizing η and minimizing $(\vec{u}.\vec{P} + w)$ can be achieved simultaneously by finding a lexicographically minimal solution to the ILP system with η', $\vec{u}$ and w in the leading position. If $\vec{u} = (u_1, u_2, \ldots, u_p)$, then the objective is as follows:

$$minimize_{\prec} \{\eta', u_1, u_2, \ldots, u_p, w\}. \quad (7)$$

The value determined for $(\vec{u}.\vec{P} + w)$ using the above objective is the least upper bound obtained for the maximum component of any conflict difference $(\vec{s} - \vec{t})$ along the hyperplane normal. Consequently, the contraction modulo can be computed as being equal to $(\vec{u}.\vec{P} + w + 1)$.

Conflict satisfaction is the primary issue involved in partitioning. So, the objective gives minimization of η' precedence over that of $(\vec{u}.\vec{P} + w)$. As we shall see later, for scenarios such as the one in Fig. 2(a), this ensures that a hyperplane which satisfies all conflicts at once is given precedence over a hyperplane which leaves some conflicts unsatisfied even if the contraction modulo for the latter is smaller than that for the former.

3.5. Finding Storage Hyperplanes Iteratively

Once a storage hyperplane $\vec{\Gamma}$ has been found as described above, it is possible that there still exist some conflicts which are not satisfied by it. Before the complete modulo storage mapping can be obtained, additional hyperplanes need to be found such that, eventually, each conflict is satisfied by at least one of the hyperplanes.

Suppose that the hyperplane $\vec{\Gamma}$ has been found based on the conflict set $CS = K_1 \cup K_2 \cup \cdots \cup K_l$. The conflicts $\vec{s} \bowtie \vec{t}$ in the conflict set CS that are not satisfied by the storage hyperplane $\vec{\Gamma}$, satisfy the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0)$. Therefore, in order to find the next storage hyperplane, the conflict set should be revised to include only the unsatisfied conflicts. This can be done by adding the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0)$ to each of the l conflict polyhedra so that the new set of conflict polyhedra are:

$$K'_i = K_i \cap \{(\vec{s}, \vec{t}) \mid \vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0\}, \quad 1 \le i \le l. \quad (8)$$

Consequently, the resulting conflict set CS' is given by:

$$CS' = \cup_{i=1}^{i=l} K'_i. \quad (9)$$

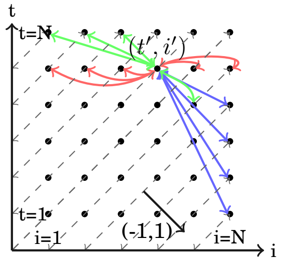


Fig. 3. Storage hyperplane $(-1, 1)$ satisfies all conflicts

In essence, instead of the original conflict set CS, the revised conflict set CS' with its constituent conflict polyhedra $K'_1, K'_2, \ldots, K'_l$, forms the basis for determining the next storage hyperplane. If all the conflicts in a conflict polyhedron K_i are satisfied by the hyperplane $\vec{\Gamma}$, its contribution to the revised conflict set CS' due to the addition of the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0)$ would be nothing. Therefore, this iterative process of determining storage hyperplanes is continued until all conflicts are satisfied, i.e., until the conflict set under consideration is empty. At each step, the contraction modulo is also found for every storage hyperplane.

Algorithm 1 summarizes the partitioning-based approach to find a modulo storage mapping. The main procedure, FIND-MODULO-MAPPING (line 1), determines the m storage hyperplanes iteratively, revising the conflict set at each step as described above (lines 4-7). The procedure, FIND-NEXT-HYPERPLANE (line 10), sets up the ILP system (lines

12-16) necessary to determine the required storage hyperplane (line 18) and the corresponding contraction modulo.

Example revisited. Consider again the producer-consumer loops which were introduced in Fig.2. In Fig.3, conflicts in different polyhedra are shown in different colors. Note that the canonical hyperplanes $(1, 0)$ and $(0, 1)$ individually do not satisfy all conflicts – the former does not satisfy the conflicts colored in red whereas the latter does not satisfy those shown in blue. However, several other hyperplanes exist which satisfy all conflicts at once e.g. $(-1, 1)$, $(-2, 1)$, $(-3, 1)$ etc. Therefore, our greedy approach would pick such hyperplanes over other candidate hyperplanes. Furthermore, the secondary objective is to minimize the contraction modulo. Among such hyperplanes which satisfy all conflicts, $(-1, 1)$ leads to the smallest contraction modulo of $2N - 1$. Since all conflicts are satisfied by the hyperplane $(-1, 1)$ itself, there is no need to find any more partitioning hyperplanes. The resulting storage mapping, $A[t, i] \to A[(i - t) \bmod (2N - 1)]$, not only reduces the dimensionality but also provides a storage size requirement which is asymptotically better than that obtained using the successive modulo technique. This modulo storage mapping is also dimension and storage optimal.

Correctness and Termination. While the primary objective is to maximize conflict satisfaction for the revised conflict set, any hyperplane which is linearly dependent on the storage hyperplanes found in previous iterations, will not satisfy any new conflict. In practice, we observed that finding the next storage hyperplane using a revised conflict set is sufficient to ensure the required linearly independence of hyperplanes. If, in addition to revising the conflict set, a theoretical guarantee for such linear independence is sought, it can be enforced by introducing additional linear independence constraints, similar to those proposed in [Bondhugula et al. 2008] for finding scheduling hyperplanes iteratively. Since the number of linearly independent storage hyperplanes required for satisfying all conflicts is at most equal to the dimensionality of the array space, the iterative process is guaranteed to terminate. A storage mapping that satisfies all conflicts is a valid one by definition: it maps conflicting indices to different partitions.

3.6. Optimality

The two-fold objective used makes our technique find good solutions which are often optimal. Note that all of the optimality discussion here is under the assumption that the mappings considered are affine. The situations where sub-optimality could creep in are as follows:

(1) In some pathological cases, a higher dimensional mapping is better than a lower dimensional one, and this may not even be known at compile time. Consider a 2-d wavefront in a 3-d iteration space with a storage mapping of size $N_1 * N_2$ versus a lower dimensional one with storage N_3. If $N_3 > N_1 * N_2$, the higher dimensional mapping leads to lower storage.
(2) Since decision variables for conflict satisfaction are added on a per conflict polyhedron basis, splitting conflict polyhedra can only yield better solutions. This is also the case when splitting dependences or iteration domains leads to better parallelization.
(3) Our first objective of conflict set satisfaction is greedy in nature and although not optimal, often finds optimal solutions in practice. Furthermore, the iterative approach to determine the partitioning hyperplanes (Section 3.5) is open to easy customization and variation – for example to enumerate a fixed number of good

```
// horizontal blur
for (x=0;x<=N−1;++x)
  for (y=0;y<=N−1;++y)
    blurx[x,y]=in[x,y]
              +in[x+1,y]
              +in[x+2,y];
// vertical blur
for (x=0;x<=N−1;++x)
  for (y=2;y<=N−1;++y)
    out[x,y]=blurx[x,y]
            +blurx[x,y−1]
            +blurx[x,y−2];
```

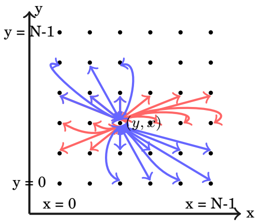
(a) 2-stage blur filter

```
for (y=0;y<=N−1;++y)
  for (x=0;x<=N−1;++x) {
/*S*/ blurx[y,x]=in[x,y]
                +in[x+1,y]
                +in[x+2,y];
      if (y>=2)
/*T*/   out[x,y]=blurx[y−2,x]
                +blurx[y−1,x]
                +blurx[y,x];
  }
```

(b) Interleaved schedule

$$\begin{aligned} CS = (((y,x),(y',x') \in blurx) \\ \vee(y'-y <= 1 \wedge y'-y >= 0 \wedge x < x') \\ \vee(y'-y <= 2 \wedge y'-y >= 1 \wedge x >= x')) \end{aligned}$$

(c) Conflict set specification



(d)

Fig. 4. The code in 4(a) and 4(b) are different versions of blur filter. The geometrical representation on the right shows the conflicts associated with the index (y, x) due to the interleaved schedule. Note that (y, x) does not conflict with the index $(y + 2, x + 1)$.

solutions and pick the minimum storage one among them, given that our storage mapping determination is quite fast (Table I).

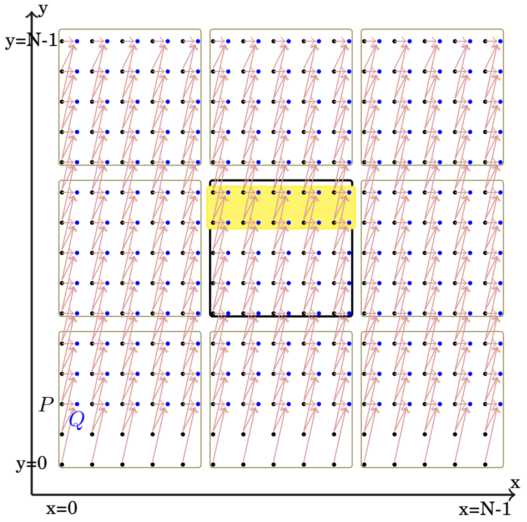
Furthermore, if there exists a single storage hyperplane that satisfies all conflicts, our approach obviously guarantees that it will be found (minimum η' value in (7)). The secondary objective ensures that the storage requirements of such a 1-dimensional modulo storage mapping found will be optimal.

4. EXAMPLES

This section discusses storage mappings obtained by our technique on several example classes of affine loop nests.

Blur filter - interleaved schedule. In image processing pipelines, such as Harris corner detectors([Harris and Stephens 1988]) instead of time-iterated stencils, a pipeline stage may apply a particular stencil once, before propagating the computed output to the next stage, which may apply a different stencil on its input. The importance of storage optimizations in domain specific compilers for image processing pipelines was studied by [Ragan-Kelley et al. 2013] for their work on the Halide DSL compiler. Consider the loop-nest of a 2-stage blur (in Fig. 4(a)). The producer-consumer locality is quite poor. It can be improved by interleaving the horizontal and vertical blurs as shown in Fig.4(b). As each statement instance $S(y, x)$ writes to $blurx[y, x]$ in accordance with the schedule $\theta(S(y, x)) = (y, x, 0)$, the last use of the value $blurx[y, x]$ is in $T(y + 2, x)$ at $\theta(T(y + 2, x)) = (y + 2, x, 1)$. The conflict set CS of the conflicting indices $(y, x) \bowtie (y', x')$ for the array space $blurx$ due to such a schedule is specified in Fig.4(c). The modulo storage mapping used by [Ragan-Kelley et al. 2013] is same as that obtained using the successive modulo technique — $blurx[y, x] \rightarrow blurx[y \bmod 3, x \bmod N]$. Fig.4(d) shows that the storage hyperplane $(-1, 2)$ would satisfy all the conflicts by itself. Furthermore, since it leads to the smallest contraction modulo of $2N + 1$, the modulo storage mapping obtained using our technique is $blurx[y, x] \rightarrow blurx[(-y + 2x) \bmod (2N + 1)]$.

Blur filter - tiled execution. Fig.5 shows a tiled version of the blur filter code introduced in Fig.4(a). The schedules for the statements P and Q can be expressed as $\theta(P(tx, ty, x, y)) = (tx, ty, x, 0, y)$ and $\theta(Q(tx, ty, x, y)) = (tx, ty, x, 1, y)$. The column-wise processing is interleaved to further improve locality so that a column of $blurx$ within the tile, once computed, is immediately read for the vertical blur along the



(a) Tiled execution of blur filter. Live-out data for the tile in yellow. The black and blue dots represent instances of the statements P and Q respectively.

```
#define in(x,y)  in[tx*B+x,ty*B+y]
#define blurx(x,y)  blurx[tx*B+x,ty*B+y]
#define out(x,y)  out[tx*B+x,ty*B+y]
for(ty=0;ty<=(N−1)/B;++ty)
    for(tx=0;tx<=(N−1)/B;++tx)
        for(x=0;x<=B−1;++x){
            for(y=0;y<=B−1;++y)
/*  P */        blurx(x,y)=in(x,y)+in((x+1),y)+in((x+2),y);
            for(y=0;y<=B−1;++y)
                if(ty*B+y>=2)
/*  Q */            out(x,y)=blurx(x,y)+blurx(x,(y−1))
                                                 +blurx(x,(y−2));
        }
// As P is enclosed by 4 loops, on total expansion of array
// blurx, the write access to blurx becomes a 4−d access
// #define blurx(x,y) (y>0)?A[tx,ty,x,y]:A[tx,ty−1,x,B+y]
```

(b) tx and ty are the tile iterators whereas x and y are the intra-tile iterators. B is the tile size.

$$((x = x') \wedge (y' > y) \wedge (tx, ty, x, y), (tx, ty, x', y') \in A)$$
$$\vee((x > x') \wedge (y' \le B - 1) \wedge (y' \ge B - 2) \wedge (tx, ty, x, y), (tx, ty, x', y') \in A)$$

(c) The geometrical representation of the conflict set is on the right.

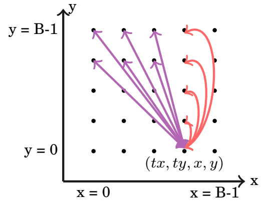
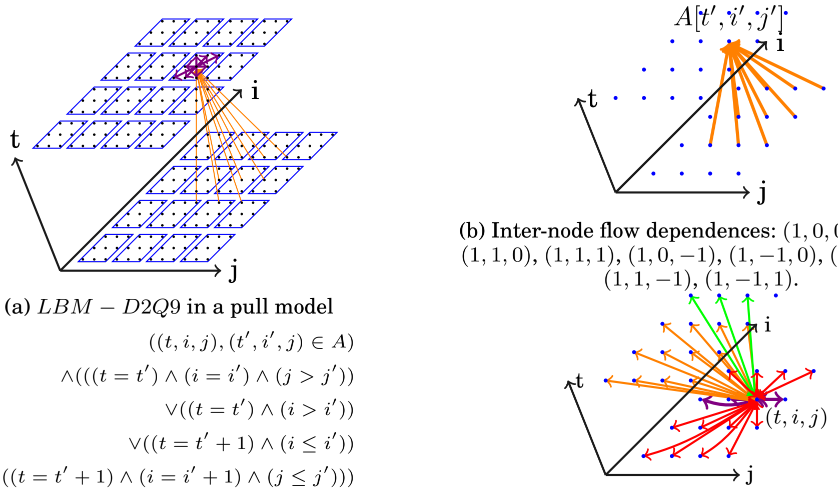


Fig. 5. A geometrical representation of the tiled execution of blur filter is show in Fig.5(a). The conflict set specification in Fig.5(c) shows all conflicts for index (tx, ty, x, y) within the same data tile. The first disjunct in the conflict set is for indices conflicting with all other indices on the same column. The conflicts due to the live-out data values which are already computed are specified by the other disjunct.

same column. The top two rows of each data tile of $blurx$ constitute its live-out data for such a schedule (refer 5(a)). Prior to contraction, a total expansion of the array space written to by the statement P is performed. This changes the write access to a 4-d access on an array space A, which has the same size and shape as the iteration domain of statement P (refer Fig.5(b)). For brevity's sake, we only consider the problem of contracting a data tile of the array space. The intra-tile conflict set is shown in Fig.5(c). The storage mapping obtained using the successive modulo technique $A[tx, ty, x, y] \rightarrow A[tx, ty, x \bmod B, y \bmod B]$ does not contract the tile at all. However, note that the hyperplane $(-2, 1)$ satisfies all the conflicts with the contraction modulo $(3B - 2)$. Our technique would arrive at this storage hyperplane to give the dimension and storage optimal storage mapping $A[tx, ty, x, y] \rightarrow A[tx, ty, (y - 2x) \bmod (3B - 2)]$. The storage required for the tile is thus reduced from B^2 down to $(3B - 2)$. Also, the same storage mapping holds even if tiles along the same row are executed in parallel.

Lattice-Boltzmann Method (LBM). is a discrete form of the Boltzmann equation [Succi 2001]. It is used in computational fluid dynamics to model complex fluid flows. An LBM kernel is characterized as $DmQn$ where m is the dimensionality of the space lattice and n is the number of particle distribution equations that need to be solved. Fig.6(a) shows a $D2Q9$ lattice arrangement. Each blue box encapsulates the solutions of 9 particle distribution equations (the 9 black dots) for a particular particle being displaced through the 2-d space lattice. The neighborhood interactions in the $D2Q9$ example are such that if all the points in every blue box are collapsed into a single node representing all the associated computations, the flow dependences are similar



(a) $LBM - D2Q9$ in a pull model

(b) Inter-node flow dependences: $(1,0,0)$, $(1,0,1)$, $(1,1,0)$, $(1,1,1)$, $(1,0,-1)$, $(1,-1,0)$, $(1,-1,-1)$, $(1,1,-1)$, $(1,-1,1)$.

$$\begin{aligned}&((t,i,j),(t',i',j') \in A)\\ &\wedge(((t=t') \wedge (i=i') \wedge (j>j'))\\ &\vee((t=t') \wedge (i>i'))\\ &\vee((t=t'+1) \wedge (i \leq i'))\\ &\vee((t=t'+1) \wedge (i=i'+1) \wedge (j \leq j')))\end{aligned}$$

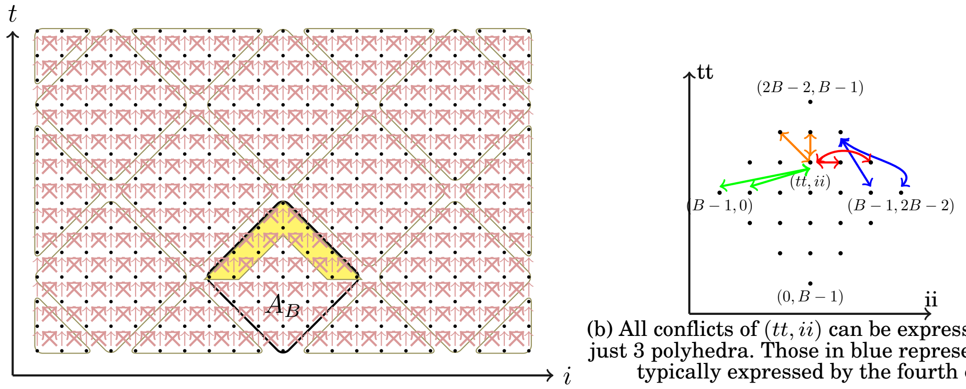
(c) The conflict set specification.

(d) Different colours differentiate conflicts from different conflict polyhedra.

Fig. 6. As in a stencil, node $A[t',i',j']$, in Fig.6(b), depends on neighboring nodes from the previous time step.

to those of a typical stencil computation (refer Fig.6(b)). Suppose that the computations associated with a node $A[t,i,j]$ are performed at logical time (t,i,j) and that the size of the array space is N in each dimension. The conflict set CS of conflicting indices $(t,i,j) \bowtie (t',i',j')$ for the array space A is specified in 6(c). There is no hyperplane which satisfies all the conflicts on its own (refer Fig.6(d)). The canonical hyperplanes $(1,0,0)$, $(0,1,0)$,$(0,0,1)$ individually do not satisfy all conflicts either. Among the hyperplanes which satisfy all conflicts in three of the four conflict polyhedra, $(-2,1,0)$ leads to the smallest contraction modulo of $(N+2)$. It satisfies all but the conflicts in violet. Since all the conflicts are not satisfied yet, another storage hyperplane must be found. The revised conflict set, containing only the unsatisfied conflicts, is essentially the conflict polyhedron made up of the violet conflicts. The hyperplane $(0,0,1)$ satisfies all of them with the smallest contraction modulo of N. The final storage mapping obtained is $A[t,i,j] \to A[(i-2t) \bmod (N+2), j \bmod N]$. The storage requirement of this 2-d mapping, (N^2+2N) is marginally more than optimal storage size of (N^2+N+1), i.e., the maximum number of live values at any point during the schedule.

Diamond tiling. [Bandishti et al. 2012] developed a technique to obtain a diamond tiling (refer Fig.7), thereby enabling the concurrent start of tiles. Let (tt,ii) be the intra-tile iterators. Suppose that the intra-tile schedule is sequential so that the value written to the memory cell $A_B[tt,ii]$ is computed at time (tt,ii). If so, the live-out portion of the data tile is as shown in Fig.7(a). The intra-tile conflicts, in accordance with the conflict set specification CS of conflicting indices $(tt,ii) \bowtie (tt',ii')$, are as shown in Fig. 7(b). Our technique arrives at the hyperplane $(1,-3)$ as it satisfies all the conflicts by itself with the contraction modulo $(6B-5)$. The final storage mapping $A_B[tt,ii] \to A_B[(tt-3*ii) \bmod (6B-5)]$ is not only dimensional optimal but also has an asymptotically better storage requirement than that of $A_B[tt,ii] \to A_B[tt \bmod B, ii \bmod (2B-1)]$, which is found using successive modulo technique.



(a) A_B is the data tile written to by iterations within the tile outlined in black. Live-out data in yellow.

$$CS = ((ii \geq ii' + 1) \wedge (tt \geq tt' + 1) \wedge (tt', ii') \in L_1)$$
$$\vee((tt \geq tt' + 2) \wedge (tt', ii') \in L_2)$$
$$\vee((tt = tt') \wedge (ii' \geq ii + 1)) \vee ((tt' = tt + 1) \wedge (ii' \leq ii))$$

(b) All conflicts of (tt, ii) can be expressed using just 3 polyhedra. Those in blue represent those typically expressed by the fourth one.

$$L_1 = \{(tt, ii) \mid (tt \leq 2B - 2) \wedge (tt \geq B - 1)$$
$$\wedge (ii \geq tt - B + 1) \wedge (ii \leq tt - B + 2)\},$$
$$L_2 = \{(tt, ii) \mid (tt \leq 2B - 2) \wedge (tt \geq B - 1)$$
$$\wedge (ii \leq B - tt - 1) \wedge (ii \geq B - tt - 2)\}$$

Fig. 7. Fig.7(a) shows diamond tiling for a stencil with flow dependences $(1, -1)$, $(1, 0)$ and $(1, 1)$. If B is the tile size, the live-out set is a union of polyhedra L_1 and L_2. The last two disjuncts in the conflict set CS represent conflicts due to flow dependences. The first two specify additional conflicts due to live-out values.

5. IMPLEMENTATION AND PRACTICAL IMPACT

We implemented the approach proposed into an automatic storage optimizer, SMO, using ISL [Verdoolaege 2010] (version isl 0.12.2) with GLPK (GNU Linear Programming kit) [GNU 2010] version 4.45 as the ILP solver. SMO accepts a conflict set specification for an array space as the input and determines a modulo storage mapping using our technique. Table I shows the storage mappings obtained for various benchmarks, and the time taken to find them (SMO time) on an Intel Core i5 2540M CPU running at 2.60 GHz. Note that in all cases where we perform tiling, tile sizes are fixed at compile time - the factor B in Table I and elsewhere is only to clarify the relationship between storage reduction and tile size.

For an n-dimensional array space, at most n linearly independent storage hyperplanes need to be found. Finding each storage hyperplane involves Fourier-Motzkin elimination to get rid of the Farkas' multipliers. Furthermore, we rely on integer linear programming to determine a storage hyperplane. Although these techniques are of exponential complexity in the worst case scenario, the compile time numbers in Table 1 (SMO time) demonstrate that they are very fast in practice.

Suppose the tiling hyperplanes for the stencil in Fig. 7 were $(1, 0)$,$(1, 1)$. Intra-tile storage optimization for such a parallelogram shaped tile with pipelined start-up would yield the storage mapping $A_B[tt, ii] \rightarrow A_B[(tt - ii) \bmod (3B - 2)$. Similarly, for a hexagonal tile [Grosser et al. 2014], the storage mapping obtained is $A_B[tt, ii] \rightarrow A_B[(-tt + 3 * ii) \bmod (9B - 7)$.

Access expression simplification. The form of our mapping is the same as in any other successive modulo optimization technique — so we do not introduce any more modulo expressions than previous ones. In fact, since our technique reduces the dimensionality of the storage mapping better than previous techniques (for eg. blur and Harris corner detection benchmarks), we will have fewer modulo expressions. Note that any potential slowdown due to the modulo expression is avoided in several cases

Table I. Storage reduction obtained using our approach (SMO) compared to the baseline successive modulo technique ([Lefebvre and Feautrier 1998]) with B being the loop blocking factor

Benchmark		Modulo storage mapping	Reduction	SMO
			(approx.)	time
produce-consume (Fig.2)	baseline	$A[t \bmod N, i \bmod N]$	$\frac{N}{2}$	
	SMO	$A[(i-t) \bmod (2N-1)]$		0.17s
blur-interleaved (Fig.4)	baseline	$blurx[y \bmod 3, x \bmod N]$	1.5	
	SMO	$blurx[(2x-y) \bmod (2N+1)]$		0.14s
blur-tiled (Fig.5)	baseline	$A[tx, ty, x \bmod B, y \bmod B]$	$\frac{B}{3}$	
	SMO	$A[tx, ty, (y-2x) \bmod (3B-2)]$		0.11s
harris-corner-tiled	baseline	$sobel[tx, ty, x \bmod B, y \bmod B]$	$\frac{B}{3}$	
	SMO	$sobel[tx, ty, (y-2x) \bmod (3B-2)]$		0.12s
unsharp-mask-tiled	baseline	$A[z, tx, ty, x \bmod B, y \bmod B]$	$\frac{B}{5}$	
	SMO	$A[z, tx, ty, (y-4x) \bmod (5B-4)]$		0.82s
LBM-D2Q9 (Fig.6)	baseline	$A[t \bmod 2, i \bmod N, j \bmod N]$	2	
	SMO	$A[(i-2t) \bmod (N+2), j \bmod N]$		0.61s
LBM-D3Q19	baseline	$A[t \bmod 2, i \bmod N, j \bmod N, k \bmod N]$	2	
	SMO	$A[(i-2t) \bmod (N+2), j \bmod N, k \bmod N]$		3.32s
LBM-D3Q27	baseline	$A[t \bmod 2, i \bmod N, j \bmod N, k \bmod N]$	2	
	SMO	$A[(i-2t) \bmod (N+2), j \bmod N, k \bmod N]$		3.33s
diamond-tile (Fig.7)	baseline	$A_B[tt \bmod B, ii \bmod (2B-1)]$	$\frac{B}{3}$	
	SMO	$A_B[(tt-3ii) \bmod (6B-5)]$		0.44s
stencil-1d-llelogram-tile	baseline	$A_B[tt \bmod B, ii \bmod B]$	$\frac{B}{3}$	
	SMO	$A_B[(tt-ii) \bmod (3B-2)]$		0.29s
stencil-1d-hex-tile	baseline	$A_B[tt \bmod B, ii \bmod (3B-2)]$	$\frac{B}{3}$	
	SMO	$A_B[(-tt+3ii) \bmod (9B-7)]$		1.15s

Table II. Performance of various benchmarks with the storage mappings shown in Table I

Benchmark	Input size	Execution time		Speedup	Storage
		baseline	smo		reduction
blur-interleaved (Fig.4)	4096×4096	0.485s	0.494s	0.98×	1.50×
blur-tiled (Fig.5)	8192×8192, B=128	0.063s	0.045s	1.40×	42.89×
unsharp-mask-tiled	4096×4096, B=512	0.645s	0.626s	1.03×	102.6×
harris-corner-tiled	8192×8192, B=64	0.710s	0.659s	1.08×	21.56×
LBM-D2Q9 (Fig.6)	1024×1024, T=500	15.05s	18.15s	0.83×	2.00×
LBM-D3Q19	200×200×200, T=100	95.68s	85.34s	1.12×	2.00×
LBM-D3Q27	200×200×200, T=100	107.8s	129.6s	0.84×	2.00×
diamond-tile (Fig. 7)	N=T=8192, B=512	1.721s	1.488s	1.16×	170.8×
stencil-1d-llelogram-tile	N=T=8192, B=8	1.478s	1.472s	1.00×	2.91×

due to the finite bounds on the affine accesses. If an access expression $(y-2x)$ ranges from say, $-2B+2$ to $B-1$, subtracting a base function from the access eliminates the modulo e.g. $A[(y-2x) \bmod 3B-2]$ can be converted to $A[(y-2x+2B-2)]$. Additionally, if the modulo factor is a power of two or just less than it, the modulo expression can be replaced with a mere bit-wise left shift. The additional arithmetic operations introduced due to the optimized storage mappings are simple integer ones. Many of the expressions are also invariant with respect to the innermost loop. Such integer arithmetic is well hidden in the pipeline, which is good for performance, but obfuscates performance analysis (its effects in isolation cannot be accurately characterized).

Impact on performance. Table II gives the execution times of various benchmarks observed when the storage mappings shown in Table I were used. For the benchmarks blur-tiled, harris-corner-tiled, diamond-tile, stencil-1d-llelogram-tile, the tiles were executed in parallel using Open MP. All the benchmarks were compiled with Intel C compiler (version 15.0) with flags "-O3 -openmp" and run on all cores of an Intel Xeon E5-2680 dual-socket machine with 8 cores per socket and a total of 64 GB of non-ECC RAM. The execution times were reproducible (less than 4% variation). It can be seen that for several benchmarks, the execution times with our storage mappings were the same if not better that those with the mappings obtained using the successive modulo technique. Among the LBM benchmarks, where weak scaling is the main issue, we even obtain a 12% speedup for LBM-D3Q19. Note that, in such cases, performance is secondary, and a reduction in storage size itself has a direct positive impact in allowing the user to run a larger problem size. We also observe that a high reduction factor in storage (tens of times) does not necessarily translate to a high performance gain. This is explained by the fact that we use a locality optimized code as the starting point for memory optimization, and there are potentially diminishing returns in performance from reduction in memory footprint. Furthermore, it is not surprising that a big reduction in memory footprint does not result in a performance improvement in certain cases. If the code is tiled for the L2 cache, storage optimization alone may not further reduce stalls due to loads and stores (as cache miss rates would have already been low). As an analogy, reducing the memory footprint say from 40 GB to 400 MB, an application may still remain compute or memory bandwidth-bound as the case may be. On the other hand, the application workload will now scale 100x with respect to data on the same hardware.

6. RELATED WORK

Storage optimization may be intra-array or inter-array depending on whether locations from the same array are reused subsequently or locations from a different array. Our approach, like all previous works discussed here, is an intra-array one. The graph coloring technique prescribed for inter-array reuse in [Lefebvre and Feautrier 1998] is complementary to intra-array approaches, and can be used in conjunction to reduce the total number of arrays used in the program.

The intra-array optimization strategy of [de Greef et al. 1997] relies on the existence of a linearized schedule θ, and on canonical linearizations of the array space. The reuse distance is computed as the maximum of the address differences between memory cells which are simultaneously live at any point during the entire execution schedule plus 1. A 1-d modulo storage mapping is obtained with the linearized access modulo the reuse distance. However, for the example in Fig. 2, even with a linearized schedule of $\theta(t, i) = tN + i$, it can be seen that this technique would not match the optimal solution found using our technique. [Quilleré and Rajopadhye 2000] consider projective memory allocation functions to optimize memory usage in ALPHA programs. The problem model used does not accommodate scenarios where a portion of the variable is live-out, for example, in the case of tiled programs or the introductory example (Fig. 1). As we have seen, this can introduce additional conflicts that do not arise due to flow dependences. Thus, for the benchmarks in Table I or Fig. 1, it will be unable to improve mappings found by the successive modulo technique.

The notion of a conflict polyhedron was introduced by [Darte et al. 2005; Alias et al. 2007] in their work on lattice-based memory allocation. The bounds and heuristics explored by [Darte et al. 2005] are under the assumption that the conflicting index difference set DS is approximated as a 0-symmetric convex polyhedron. Our approach is fundamentally different – relying on the notion of *conflict satisfaction*, and works naturally with the conflict set expressed as a union of polyhedra. As we have seen, for tiled codes with boundary live-outs in multiple directions, our approach leads to an order of magnitude reduction in storage; this is a drastic reduction in memory, immediately observable in common practical cases. For the simple producer-consumer example in Fig. 2, with $n = 9$, all heuristics implemented in Bee+cl@k [Alias 2007] determined the same modulo storage mapping of $a[t, i] \to a[t \bmod 9, i \bmod 9]$, using the same bases for computing the contraction moduli as that suggested by [Lefebvre and Feautrier 1998]. Their optimal search-based method for the convex approximation of the problem came up with the mapping, $a[t, i] \to a[t \bmod 1, (14t + i) \bmod 61]$, which clearly uses more storage than $a[t, i] \to a[(i - t) \bmod 17]$, obtained using our technique.

[Strout et al. 1998] introduced the concept of an occupancy vector which captures the duration (as a distance vector) after which a location can be reused in a repeated fashion. The universal occupancy vector (UOV) is one which is valid for any valid loop schedule, and a search-based approach is proposed to find the optimal UOV. The storage mappings that they derive from a UOV and our storage mappings are conceptually similar in that both specify an array partitioning, but differ in their mathematical form and in the approach used to find them. Since our mappings are for a particular schedule, they are expected to lead to less storage: for eg., assuming identity schedules, $(N + 3)$ versus $2N$ for a 5-point 1-d stencil, $N^2 + 2N$ instead of $2N^2$ for LBM-D2Q9. However, schedule-independent UOV-based solutions for programs with constant dependences and multi-boundary live-outs (tiled or untiled) are still an order of magnitude better than those which use canonical bases [Lefebvre and Feautrier 1998], and those which are unable to find the right bases for such codes [Quilleré and Rajopadhye 2000; Darte et al. 2005; Alias et al. 2007]. Another key difference is that the approach in [Strout et al. 1998] is designed for perfect loop nests with constant dependences. [Thies et al. 2001; Thies et al. 2007] extend the notion of occupancy vector to an affine occupany vector which is valid for any valid affine schedule. They also discuss a technique for determining a storage mapping given an affine schedule. However, the given schedule needs to be one-dimensional restricting the class of programs which lend themselves to their technique. In contrast, our technique supports multi-dimensional schedules even as it contracts the array along multiple dimensions.

Although there is complex interplay between a schedule and storage optimization, schedules have a direct impact on other important aspects, evidently parallelism and single-thread performance. The overall scheme that our approach fits in is thus one of first determining a schedule and then reducing its memory footprint maximally.

7. CONCLUSION

Automatic solutions to the intra-array storage optimization problem are crucial for high-level and domain-specific language compilers. We cast the problem as one of array space partitioning where each partition uses the same memory location. This allowed us to develop an algorithm to find the right orientations for the array partitioning hyperplanes. The algorithm handles non-convex conflict relations described as union of polyhedra, and is driven by the two objectives of maximizing conflict satisfaction and minimizing conflict distances. For numerous examples and real-world problems, we showed significant reductions in storage requirement over previous techniques, ranging from a constant factor to asymptotic in loop blocking factor or array extents – the latter being a dramatic improvement for practical purposes. We showed that our technique is dimension- and size-optimal if a one-dimensional storage mapping exists, and that it often finds optimal affine mappings in practice.

REFERENCES

Samah Abu-Mahmeed, Cheryl McCosh, Zoran Budimli, Ken Kennedy, Kaushik Ravindran, Kevin Hogan, Paul Austin, Steve Rogers, and Jacob Kornerup. 2009. Scheduling Tasks to Maximize usage of Aggregate Variables In Place. In *International Conference on Compiler Construction (CC)*.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. 2006. *Compilers: Principles, Techniques, and Tools Second Edition*. Prentice Hall.

Christophe Alias. 2007. Bee+Cl@k. (2007). Bee+Cl@k tool: http://compsys-tools.ens-lyon.fr/bee/.

Christophe Alias, Fabrice Baray, and Alain Darte. 2007. Bee+Cl@k: an implementation of lattice-based array contraction in the source-to-source translator Rose. In *Languages Compilers and Tools for Embedded Systems*. 73–82.

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC*. Article 40, 11 pages.

Uday Bondhugula, M. Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*.

Alain Darte, Robert Schreiber, and Gilles Villard. 2005. Lattice-Based Memory Allocation. *IEEE Trans. Comput.* 54, 10 (2005), 1242–1257.

Eddy de Greef, Francky Catthoor, and Hugo De Man. 1997. Memory Size Reduction Through Storage Order Optimization for Embedded Parallel Multimedia Applications. *Parallel Comput.* 23, 12 (1997), 1811–1837.

P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem: Part I, One-dimensional time. *International Journal of Parallel Programming* 21, 5 (1992), 313–348.

GNU. 2010. GLPK (GNU Linear Programming Kit). (2010). https://www.gnu.org/software/glpk/.

Tobias Grosser, Albert Cohen, Justin Holewinski, P Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 66.

Chris Harris and Mike Stephens. 1988. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*. 147–151.

Vincent Lefebvre and Paul Feautrier. 1998. Automatic Storage Management for Parallel Programs. *Parallel Comput.* 24, 3-4 (1998), 649–671.

Fabien Quilleré and Sanjay V. Rajopadhye. 2000. Optimizing memory usage in the polyhedral model. *ACM Trans. Program. Lang. Syst.* 22, 5 (2000), 773–815.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. 519–530.

Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons.

M. Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. 1998. Schedule-Independent Storage Mapping for Loops. In *International conference on Architectural Support for Programming Languages and Operating Systems*. 24–33.

S. Succi. 2001. *The Lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press.

William Thies, Frédéric Vivien, and Saman Amarasinghe. 2007. A Step Towards Unifying Schedule and Storage Optimization. *ACM Trans. Program. Lang. Syst.* 29, 6, Article 34 (Oct. 2007). DOI:http://dx.doi.org/10.1145/1286821.1286825

William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman P. Amarasinghe. 2001. A Unified Framework for Schedule and Storage Optimization.. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*. 232–242.

Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Vol. 6327. Springer, 299–302.