SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization

Somashekaracharya Bhaskaracharya, Uday Bondhugula, Albert Cohen

SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization

Somashekaracharya G. Bhaskaracharya
Department of CSA
Indian Institute of Science
Bangalore 560012 India
& National Instruments, Bangalore
gbs@csa.iisc.ernet.in

Uday Bondhugula
Department of CSA
Indian Institute of Science
Bangalore 560012 India
uday@csa.iisc.ernet.in

Albert Cohen
INRIA and DI, École Normale Supérieure
Paris 75005 France
albert.cohen@inria.fr

Abstract

The polyhedral model provides an expressive intermediate representation that is convenient for the analysis and subsequent transformation of affine loop nests. Several heuristics exist for achieving complex program transformations in this model. However, there is also considerable scope to utilize this model to tackle the problem of automatic memory footprint optimization. In this paper, we present a new automatic storage optimization technique which can be used to achieve both intra-array as well as inter-array storage reuse with a pre-determined schedule for the computation. Our approach works by finding statement-wise storage partitioning hyperplanes that partition a unified global array space so that values with overlapping live ranges are not mapped to the same partition. Our heuristic is driven by a fourfold objective function which not only minimizes the dimensionality and storage requirements of arrays required for each high-level statement, but also maximizes inter-statement storage reuse. The storage mappings obtained using our heuristic can be asymptotically better than those obtained by any existing technique. We implement our technique and demonstrate its practical impact by evaluating its effectiveness on several benchmarks chosen from the domains of image processing, stencil computations, and high-performance computing.

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

Keywords Compilers, storage mapping optimization, memory optimization, array contraction, polyhedral framework

1. Introduction and Motivation

Programs that use arrays often provide significant opportunity to minimize storage requirement by reusing memory locations. The reuse may be across elements of a single array or across multiple arrays. Intra-array storage management deals with how an array written to by a particular statement is compacted and accessed. Inter-array reuse analysis pertains to memory locations from different arrays being written to in different high-level statements of a program. It might be possible to reduce the number of arrays to minimize the amount of allocated storage. Consider a high-level statement which writes to an array appearing within an arbitrarily nested loop. Multiple dynamic instances of the statement (arising out of an outer surrounding loop) can store values that they compute to the same memory location provided the lifetimes of these values do not overlap. Besides general-purpose programming languages, storage optimization assumes high importance in domain-specific languages where high-level constructs provided to the programmer abstract away storage—providing the compiler with complete freedom in allocating and managing storage [1, 11, 13, 15].

Consider the code in Figure 1(a). It uses two buffers `P` and `Q` in a ping-pong fashion so that the updated values are not immediately used in the same time (t) loop iteration. Such a pattern is common to several Jacobi-style smoothing operations used in iterative solution of partial differential equations, and in other stencil computations used in image processing. It is not obvious whether the total storage requirement of $2N$ (N for each of the two arrays) can be reduced any further, and developers of such code often assume that two arrays are needed. State-of-the-art intra-array storage optimization techniques and heuristics like that of Lefebvre and Feautrier [12], Darte et al. [5], and Bhaskaracharya et al. [4] use modulo mappings to compact storage—the introduction of a modulo operator in the access expression leads to reuse of memory within the same array. In this case, if one analyzes intra-array lifetimes, no modulus smaller than N can be deduced—this effectively means no storage can be compacted. On the other hand, inter-array reuse techniques that analyze and capture the interference of lifetimes of different arrays are again unable to reuse storage between `P` and `Q`—this is because the lifetimes of both arrays are interleaved, and techniques like those based on graph coloring [12] or that of De Greef et al. [10] will be unable to reduce storage any further. Hence, no existing automatic intra-array or inter-array storage optimization technique can further optimize memory for the code in Figure 1(a).

However, a framework that takes an integrated and precise view of intra and inter-array memory reuse can indeed reduce storage from 2N to N+1. A mapping that enables this compaction is given by:

$$\mathtt{P_t[i]} \rightarrow \mathtt{A[(i - t + N)\%(N + 1)]}$$
$$\mathtt{Q_t[i]} \rightarrow \mathtt{A[(i - t + N)\%(N + 1)]}$$

where $P_t[i]$ and $Q_t[i]$ represent the values $P[i]$ and $Q[i]$ computed in iteration t of the outermost loop. Such a mapping leads to the

```
for (t=1; t<=N; t++){
  for (i=1; i<=N; i++)
    P[i] = f(Q[i-1],Q[i],Q[i+1]);
  for (i=1; i<=N; i++)
    Q[i] = P[i];
}
for (i=1;i<=N;i++)
  result +=Q[N][i];
```

(a) A stencil using a ping-pong buffer

```
for (t=1; t<=N; t++){
  for (i=1; i<=N; i++)
    A[(i-t+N)%(N+1)] = f(A[(i-t+N)%(N+1)],
      A[(i-t+1+N)%(N+1)], A[(i-t+2+N)%(N+1)]);
  for (i=1; i<=N; i++)
    A[(i-t+N)%(N+1)] = A[(i-t+N)%(N+1)];
}
for (i=1;i<=N;i++)
  result +=A[i%(N+1)];
```

(b) Fig.1(a) with an optimized storage mapping

```
for (t=1; t<=N; t++)
  for (i=1; i<=N; i++)
    A[(i-t+N)%(N+1)] = f(A[(i-t+N)%(N+1)],
      A[(i-t+1+N)%(N+1)], A[(i-t+2+N)%(N+1)]);
for (i=1;i<=N;i++)
  result +=A[i%(N+1)];
```

(c) After elimination of the dead code in Fig.1(b)

Figure 1. Storage optimization of a 1-d stencil using a ping-pong buffer (from 2N to N+1)

code shown in Figure 1(b) with storage of just N+1 for array `A`. A surprising and positive side-effect of this mapping is that the second statement is turned into dead code! A subsequent compiler pass can completely eliminate the second statement. This optimization opportunity is non-trivial to infer from the original code. Not only does such a mapping lead to smaller storage, but also eliminates an unnecessary copy between the arrays while preserving semantics. The approach we present in this paper is able to determine such mappings automatically. In the case of more realistic examples which employ 2-d or 3-d arrays, the reduction is more prominent: from $2N^2$ to $N^2 + 2N$ for a code similar to Fig. 1(a) that uses 2-d arrays, and from $2N^3$ to about $N^3 + 2N^2$ for one employing 3-d arrays. This allows a programmer to effectively process larger problems given a fixed amount of main memory available on a system, and use fewer physical nodes to solve a problem of a given size.

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 [2].

Our approach builds on the intra-array storage optimization work of Bhaskaracharya et al. [4] that introduced the notion of *storage hyperplanes*. A storage hyperplane defines a partitioning of the iteration space such that each partition writes to the same memory location. The approach is then of iteratively finding a minimum number of *storage hyperplanes* with certain objectives. 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. For example, the hyperplane that enables the storage optimization in Figure 1(b) is $(i-t) = (-1, 1) \cdot (t, i)^T$. Unlike in Bhaskaracharya et al. [4], storage hyperplanes in this work have a meaning not just within an array but also across arrays. Often, programs intensive in data are written with arrays being produced as outputs while being consumed subsequently. The full extent of storage optimization can only be performed with a global view of conflicts.

Our integrated approach to memory optimization subsumes previous intra-array optimization approaches while allowing effective inter-array optimization. The framework is also more powerful than an approach that decouples the two problems and solves them separately. In summary, our contributions are the following.

- We describe an integrated approach to intra-statement as well as inter-statement storage optimization while casting this as a problem of partitioning a unified global array space; each partition created corresponds to the same memory location.
- We then formulate an optimization problem solvable using a greedy heuristic whose objective takes into account the dimensionality of the mapping, the extents along each dimension, and inter-statement storage reuse considerations.
- We implement and evaluate our technique on various domain-specific benchmarks and demonstrate significant reductions in storage requirement—ranging from a constant factor to asymptotic in the extents of the original array dimensions or loop blocking factors.

Section 2 provides the necessary background on the framework used, and introduces the notation used later. Section 3 and 4 describe our storage optimization scheme in detail. Various examples from real-world applications are discussed in Section 5. Section 6 reports results from our implementation. Related work and conclusions are presented in Section 7 and Section 8 respectively.

2. Background

This section provides the notation and background for the techniques we present in the rest of this paper. The terms storage and memory are used interchangeably herein—both refer to main memory utilization.

DEFINITION 1. *The set of all vectors $\vec{v} \in \mathbb{Z}^n$ such that $\vec{\Gamma}.\vec{v} = \delta$ constitutes an affine hyperplane.*

Different constant values for δ generate different parallel instances of the hyperplane. The orientation of a particular hyperplane is characterized by the vector $\vec{\Gamma}$, and its offset is given by δ.

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_S enclosing loops, is represented as an integer point in an n_S-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.

2.1 Successive Modulo Technique

Lefebvre and Feautrier [12] proposed an intra-array 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 S. Without any loss of generality, if

we assume that the loop indices are non-negative, then $\vec{i}_S$ writes to $A_S[\vec{i}_S]$. 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. *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 $\theta(\vec{i}) \preceq \theta(L(\vec{j}))$ and $\theta(\vec{j}) \preceq \theta(L(\vec{i}))$ 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 symmetric, non-reflexive. Partial expansion is performed iteratively with each statement being considered once at every depth of the surrounding loop-nest. The *contraction modulus* e_p (or expansion degree as Lefebvre and Feautrier 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 modulus is given by $e_p = b_p + 1$, where $b_p \hat{u}_p$ is the component of $\vec{b}$ along the axis i_p, with $\hat{u}_p$ representing the unit vector along the same axis. In essence, the contraction modulus 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 $A_S[\vec{i}_S \bmod \vec{e}]$, where $\vec{e} = (e_0, e_1, \ldots, e_{n_S - 1})$. This method to determine the contraction moduli will hereafter be referred to as the *successive modulo technique*.

2.2 Rectangular Hull for Inter-Array Reuse

Lefebvre and Feautrier [12] also propose a graph coloring based approach for inter-array reuse. In this approach, an interference graph is constructed where each node represents a statement. An edge between two nodes S_i and S_j implies that the two statements cannot write to the same data structure i.e., a rectangular hull of the arrays A_{S_i} and A_{S_j} contracted in accordance with the successive modulo technique. Such inter-array reuse is possible only if a value computed by the statement S_i is not overwritten prematurely, before its last use, by an execution instance of the statement S_j and vice versa. A greedy coloring algorithm can then be applied on such an interference graph to determine the statements which can write to such a shared data structure. Hereafter, we refer to this technique as the *rectangular hull method.*

2.3 Conflict Satisfaction and Storage Hyperplanes

Recently, Bhaskaracharya et al. [4] proposed an alternative approach to intra-array storage optimization. They cast the intra-array reuse problem as one of partitioning the array space A of a statement S through *storage partitioning hyperplanes*. The partitioning heuristic provided by them relies on the notion of satisfaction of a conflict $\vec{i} \bowtie \vec{j}$ in the conflict set CS. This notion can be formalized as follows.

DEFINITION 3. *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. So, the problem of intra-array storage optimization for a given statement S with an n_S-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 $m \times n_S$ transformation matrix constructed using the m storage hyperplanes as the m rows of the matrix. The contraction moduli computed along the normals of the m hyperplanes form the m components of the vector $\vec{e}$.

2.3.1 Analyzing Conflict Satisfaction

We now explain the approach of Bhaskaracharya et al. [4] for analyzing the conflicts represented in the conflict set CS, which can be specified as a union of convex polyhedra (also called conflict polyhedra). 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 $\vec{s}, \vec{t} \in A$ where A is the array space of the statement S. By Definition 3, the hyperplane $\vec{\Gamma}$ satisfies this conflict if $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \neq 0$. Since the array space A is bounded, there must exist finite upper bounds of the form $(\vec{u}.\vec{P} + w)$ on the conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$, where $\vec{P}$ is the vector of program parameters. Now, suppose $(\vec{u}.\vec{P} + w)$ is the upper bound on any conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$. As the conflict relation is symmetric, $(\vec{u}.\vec{P} + w)$ can be treated as an upper bound on the absolute value of the conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$. Furthermore, since $(\vec{u}.\vec{P} + w)$ is finite, there must exist a finite upper bound on it of the form $(c\vec{P} + c)$. In essence, these bounding constraints can be expressed as follows:

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

In practice, a high value such as $c = 10$ (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.

In general, the conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$ could be positive, negative, or zero. This tri-state nature of conflict satisfaction can be captured by introducing a pair of binary decision variables x_{1i}, x_{2i} for each conflict polyhedron K_i such that:

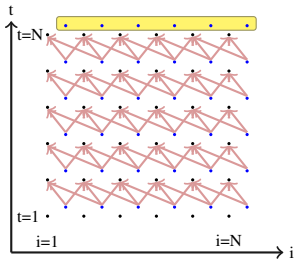
$$x_{1i} = \begin{cases} 1 & \text{if } (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1,\ \forall\ \vec{s} \bowtie \vec{t} \in K_i, \\ 0 & \text{if otherwise,} \end{cases}$$

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

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

```
// time−iterated stencil
for (t=1;t<=N;t++){
  for (i=1;i<=N;i++)
/*S0*/ A0[t][i]=f((i>1 && t>1 ? A1[t−1][i−1] : Q[i−1]),
                  (t>1 ? A1[t−1][i] : Q[i]),
                  (i<N && t>1 ? A1[t−1][i+1] : Q[i+1]));
  for (i=1;i<=N;i++)
/*S1*/ A1[t][i]=A0[t][i];
}
for (i=1;i<=N;i++)
  result +=A1[N][i];
```

(a) 1-d stencil from Fig. 1(a) after total expansion.



(b) The instances of S_1 which compute the live-out date are shown in yellow.

$$L = \{(t,i) \mid (t,i) \in A_1 \wedge (t = N)\}$$
$$CS_0 = ((t,i),(t',i') \in A_0 \wedge (t = t') \wedge (i < i'))$$
$$CS_1 = ((t,i),(t',i') \in A_1 \wedge (((t = t') \wedge (i < i'))$$

(c) The intra-statement conflict sets for Fig. 2(a).

$$CS_{0,1} = ((t,i) \in A_0 \wedge (t',i') \in A_1 \wedge (t = t') \wedge (i > i'))$$
$$CS_{1,0} = ((t,i) \in A_1 \wedge (t',i') \in A_0 \wedge (((t+1 = t') \wedge (i \geq i'))$$

(d) The inter-statement conflict sets for Fig. 2(a).

Figure 2. A geometric representation of the iteration domains of statements S_0 and S_1 in Fig. 1(a) is shown in Fig. 2(b). The black and blue dots represent instances of the statements S_0 and S_1 respectively. The maroon arrows represent the flow dependences from S_1 to S_0.

by the conjunction:

$$(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1 - (1 - x_{1i})(c\vec{P} + c + 1)$$
$$\wedge \quad (\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq -1 + (1 - x_{2i})(c\vec{P} + c + 1). \qquad (2)$$

By definition, x_{1i} and x_{2i} cannot be simultaneously equal to one. 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 a conflict satisfaction constraint: $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1$. The other conjunct is reduced to the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq (c.\vec{P} + c)$, which is implied by the bounding constraints (1). Similarly, if $x_{2i} = 1$ and $x_{1i} = 0$, the first conjunct becomes $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq -(c.\vec{P} + c)$ which is again implied by the bounding constraints (1). 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 conjunction boils down to the bounding constraint (1), which must always hold, regardless of whether all or a few of the conflicts in the conflict polyhedron are satisfied.

3. An Integrated Approach to Storage Optimization

The successive modulo technique is quite versatile, scalable and also parametric. However, the modulo storage mappings obtained for the statements are not always of minimum storage. Furthermore, the construction of array interference graphs for inter-statement storage reuse is based on a straightforward computation of the rectangular hull which can also fail to exploit the full potential for inter-statement storage reuse. Bhaskaracharya et al [4] addressed only the first of these limitations. In this section, we present the basic framework for a unified approach that can be used to exploit intra-statement as well as inter-statement storage reuse opportunities.

3.1 A Simple Example

Consider the static control loop-nest in Fig. 1(a) performing a ping-pong style 1-d stencil computation. Suppose the statements S_0 and S_1 are executed according to identity schedules: $\theta_0(t,i) = (t,0,i)$ and $\theta_1(t,i) = (t,1,i)$ respectively. Since the loop-nest is not in single-assignment form, prior to the application of the successive modulo technique, the statements are rewritten so that each statement instance $S_0(t,i)$ writes to its own distinct memory cell $A_0[t,i]$; likewise, for the statement S_1. Consequently, the array spaces A_0 and A_1 have the same size and shape as the iteration domains of the statement S_0 and S_1 respectively. Such a single-assignment version is shown in Fig. 2(a).

A few of the values computed by statement S_1 are live even after the entire loop nest has been executed. These live-out values reside in the set of memory cells L, specified in Fig. 2(c). Essentially, the top row computed by S_1 is live-out (refer Fig. 2(b)). In general, the conflict set 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. The conflict sets for the statements S_0 and S_1, CS_0 and CS_1, are made up of pairs of conflicting indices $(t,i) \bowtie (t',i')$ and can be represented as unions of convex polyhedra. The constraints for these conflict sets are as specified in Fig. 2(c). Although the conflict relation is, strictly speaking, symmetric, the constraints represent a conflict between a pair of conflicting indices only once, effectively treating it as an unordered pair. A geometrical representation of these intra-statement conflicts is shown in Fig. 3. The conflict set CS_0 specifies that each instance of the statement S_0 conflicts with all other instances of S_0 in the same row. Similarly, the conflict set CS_1 is also made of conflicts involving different indices from the same row. The constraints in CS_1 capture the conflicts created by the live-out values as well.

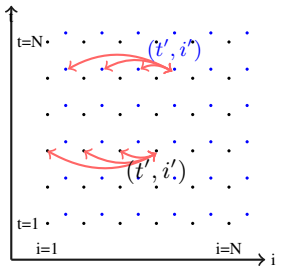


Figure 3. The red arrows denote the intra-statement conflicts (see Fig.2(c)).

3.1.1 Successive Modulo + Rectangular Hull

Applying the successive modulo technique on the conflict set CS_0, at loop-depth 0, the contraction modulus obtained is 1 as there are no conflicts along the t dimension. However, the contraction modulus at loop-depth 1 is N due to the conflict $(1,1) \bowtie (1,N)$. The resulting modulo storage mapping for the statement S_0 is $A_0[t,i] \rightarrow A[t \bmod 1, i \bmod N]$. For similar reasons, the contraction moduli for the conflict set CS_1 are also 1 and N respectively. The modulo storage mapping for the statement S_1 is therefore, $A_1[t,i] \rightarrow A[t \bmod 1, i \bmod N]$. In other words, both A_0 and A_1 are contracted to $1-$dimensional arrays of size N.

Suppose A_{0-1} is the rectangular hull of the arrays A_0 and A_1 thus contracted using the successive modulo technique. Clearly, A_{0-1} is also a $1-$dimensional array of size N. Now, instead of the contracted arrays A_0 and A_1, suppose the statements S_0 and S_1 operate on this rectangular hull A_{0-1} in accordance with

the write relations $S_0(t,i) \rightarrow A_{0-1}[t \bmod 1, i \bmod N]$ and $S_1(t,i) \rightarrow A_{0-1}[t \bmod 1, i \bmod N]$ respectively. Clearly, such a storage mapping would create an output dependence between $S_1(t,i)$ and $S_0(t+1,i)$ i.e. the value computed by $S_1(t,i)$ would be overwritten with the value computed by $S_0(t+1,i)$ prematurely, before a pending use of the former in the statement $S_0(t+1,i+1)$. Therefore, such a rectangular hull cannot be used to serve inter-statement storage reuse for the statements S_0 and S_1.

As already shown, a better storage mapping for the above example would be $A_j[t,i] \rightarrow A[(i-t) \bmod (N+1)]$ for $j = 0, 1$. Such a 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 entire loop-nest has been executed. Furthermore, it achieves both intra-statement as well as inter-statement storage reuse, while reducing the storage requirement for the loop-nest from $2N$ to $N+1$. This example shows that a straightforward computation of the contraction moduli along the canonical bases for intra-statement storage reuse, followed by a simple rectangular hull estimate for inter-statement storage reuse, can lead to solutions which can be worse than the optimal solution. As will be explained in the following sections, a better approach is to find storage hyperplanes for each statement which partition a global array space based on a global conflict set specification. The required contraction moduli can then be computed along the hyperplane normals.

3.2 A Global Array Space

The process of total data expansion, described earlier in Section.2.1, is used to ensure that each statement S_j writes to its own distinct array space A_j which has the same size and shape as the iteration domain of S_j. Suppose d_j is the dimensionality of the array space A_j, with d being the maximum dimensionality of any such array space. Consequently, the write relation for statement S_j is of the form $S_j(\vec{i}) = A_j[i_0, i_1, \ldots, i_{d_j-1}]$. Instead of creating separate arrays for each statement in this manner, we unify these arrays into a single *global array space* A of $(d+1)$ dimensions. The given program can then be translated to single-assignment form by rewriting it so that each statement S_j writes to the global array space. This must be in accordance with the write relation $S_j(\vec{i}) \rightarrow A[j, i_0, i_1, \ldots, i_{d_j-1}, 0, \ldots, 0]$ with $(d-d_j)$ trailing zeroes for indexing the $(d-d_j)$ innermost dimensions. As in the total data expansion process, the read accesses are altered accordingly to eliminate the output and anti-dependences, while respecting flow dependences. The subspace $A[j]$ in the global array space A that is written to by the statement S_j is said to constitute the *local array space* of S_j. It can be seen that $A[j]$ is nothing but A_j padded with $(d-d_j)$ additional inner dimensions.

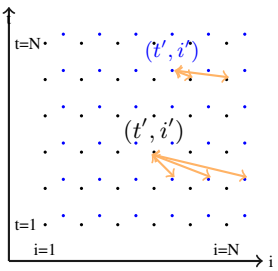


Figure 4. The orange arrows denote inter-statement conflicts (cf. Fig.2(d)).

The conflict set for exploiting intra-statement storage reuse need only consist of conflicts involving indices from the same local array space. Such conflicts are referred to as intra-array or *intra-statement conflicts*. Analogously, it is also possible to think of inter-array or *inter-statement conflicts* spanning two different local array spaces which must be analyzed in order to exploit inter-statement storage reuse. A global array space allows us to define a global conflict set that is a specification not just of the intra-statement conflicts but also of the inter-statement conflicts. The global conflict set can then serve as the basis for finding suitable storage mappings for each statement. The inter-statement conflicts for the example in Fig. 2(a) are specified in Fig. 2(d). The conflict set $CS_{0,1}$ represents conflicts which specify that a value computed by S_1 must not overwrite those computed by S_0 which still have a pending use. Similarly, the conflict set $CS_{1,0}$ specifies that a value computed by S_0 must not be overwritten prematurely by S_1. A geometrical representation of these conflicts is shown in Fig 4. The global conflict set CS for the global array space is nothing but the union of the inter-statement conflicts and intra-statement conflicts specified in Fig. 2(c) and Fig. 2(d).

3.3 Conflict Satisfaction in a Global Array Space

We formalize here the notion of storage partitioning hyperplanes (or storage hyperplanes) satisfying a conflict $\vec{i} \bowtie \vec{j}$ in the global conflict set CS. Suppose the conflict is between indices from local array spaces $A[s]$ and $A[t]$ corresponding to the statements S_s and S_t respectively. The conflict is an intra-statement one if $s = t$, otherwise it is an inter-statement conflict.

DEFINITION 4. *Given a pair of indices $\vec{i}$ and $\vec{j}$ in the global array space A such that $\vec{i} \in A[s]$ and $\vec{j} \in A[t]$, a conflict between $\vec{i}$ and $\vec{j}$ is said to be satisfied by the hyperplanes $\vec{\Gamma}_s$ and $\vec{\Gamma}_t$ with corresponding offsets δ_s and δ_t if $(\vec{\Gamma}_s.\vec{i} + \delta_s - \vec{\Gamma}_t.\vec{j} - \delta_t) \neq 0$.*

Imagine the global array space being partitioned separately by the hyperplanes Γ_s and Γ_t for the statements S_s and S_t. The conflict $\vec{i} \bowtie \vec{j}$ is said to be satisfied by them if the array indices are not mapped to the same partition. This a generalization of Definition 3 for multiple statements. Furthermore, an important difference is that we characterize a hyperplane by both its normal and its offset whereas Bhaskaracharya et al. [4] only consider the normal. The rationale behind considering the offset is that a well-chosen constant shift of the local array spaces can often enable inter-statement storage reuse. However, if the purpose is only intra-array reuse, the offset can be dispensed with.

The successive modulo technique can also be understood through this notion of conflict satisfaction. Consider again the loop-nest in Fig.2. Suppose A is the 3-dimensional global array space obtained by unifying the local array spaces A_0 and A_1. Finding the contraction moduli along the canonical axes t and i is then akin to partitioning the global array space A through the storage hyperplanes $(0,1,0)$ and $(0,0,1)$, with zero offset for both. Together, they satisfy all intra-statement conflicts. For example, if we consider the statement S_1, there are no conflicts across rows in the local array space A_1. Consequently, the hyperplane $(0,1,0)$ does not satisfy any conflict. The contraction modulus for this storage hyperplane is therefore equal to 1. However, the local array space A_1 is then divided into N partitions by the storage hyperplane $(0,0,1)$, which satisfies all the intra-statement conflicts of S_1 (none of which were satisfied by the previous storage hyperplane) e.g. $A[1,1,1] \bowtie A[1,1,N]$ is satisfied by the hyperplane $(0,0,1)$. As a result, the conflicting indices in the conflicts which were not satisfied at the previous level end up in 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. Furthermore, in accordance with the rectangular hull method for inter-statement storage reuse, the two contracted arrays cannot be fused. This can be understood as the inter-statement conflicts being satisfied by the zero-offset hyperplane $(1,0,0)$, with a contraction modulus equal to two. Consequently, the storage mappings obtained for the statements S_0 and S_1 are $A[0,t,i] \rightarrow A[t \bmod 1, i \bmod N, 0 \bmod 2]$ and $A[1,t,i] \rightarrow A[t \bmod 1, i \bmod N, 1 \bmod 2]$ respectively.

All the intra-statement and inter-statement conflicts specified in Fig.2(c) and Fig.2(d) can thus be seen as being satisfied using the three canonical hyperplanes, $(0,1,0)$, $(0,0,1)$ and $(1,0,0)$, considered in that order. The above description of the successive modulo technique and the rectangular hull method suggests that the storage hyperplanes could trivially correspond to the canonical axes of the global array space. The dimensionality of the global array space is then a loose upper bound on the number of storage hyperplanes that need to be found for a particular statement in order to satisfy all conflicts associated with it. However, the alternative mapping $A[j,t,i] \rightarrow A[(i-t) \bmod (N+1)]$ is better than the solution thus resulting for the above example not only in terms of storage size required but also in terms of the amount of inter-statement reuse. The existence of such a mapping suggests that it is possible to satisfy all the conflicts—intra-statement as well as inter-statement—using just one storage hyperplane.

3.4 A Global Array Space Partitioning Approach

Consider a static control part with r statements $S_0, S_1, \ldots, S_{r-1}$. Suppose that each statement S_j, with an n_j-dimensional iteration domain D_j, writes to an array space A_j (of the same size and shape as D_j due to total data expansion). Let A be the $n-$dimensional global array space constructed by unifying all the individual array spaces. The problem of exploiting intra-statement as well as inter-statement storage reuse can be seen as a problem of finding a set of m partitioning hyperplanes $\vec{\Gamma}_j^{(1)}, \vec{\Gamma}_j^{(2)}, \ldots, \vec{\Gamma}_j^{(m)}$ with corresponding offsets $\delta_j^{(1)}, \delta_j^{(2)}, \ldots, \delta_j^{(m)}$ for each statement S_j such that the following conditions are met.

- Every conflict within the local array space $A[j]$ of statement S_j must be satisfied by at least one of the m hyperplanes found for it.
- An inter-statement conflict involving the statements S_j and S_k must be satisfied by at least one pair of hyperplanes $\Gamma_j^{(l)}$ and $\Gamma_k^{(l)}$, both of which are at the same level l for the statements S_j and S_k respectively.

Consider the storage hyperplanes $\vec{\Gamma}_0^{(l)}, \vec{\Gamma}_1^{(l)}, \ldots, \vec{\Gamma}_{r-1}^{(l)}$ found for the r statements at a certain level l. The contraction modulus $e_j^{(l)}$ for a statement S_j at level l can be computed as the maximum conflict difference among all the conflicts associated with the statement S_j which are satisfied at that level i.e., $max(\mid \vec{\Gamma}_j^{(l)}.\vec{s} + \delta_j^{(l)} - \vec{\Gamma}_k^{(l)}.\vec{t} - \delta_k^{(l)} \mid)$ for all conflicts $\vec{s} \bowtie \vec{t}$ being satisfied by the hyperplanes $\vec{\Gamma}_j^{(l)}$ and $\vec{\Gamma}_k^{(l)}$. Therefore, the m-dimensional modulo storage mapping for each statement S_j would be of the form $A[\vec{i}] \rightarrow A[(M_j\vec{i} + \vec{\delta}_j) \bmod \vec{e}_j]$. The vector $\vec{e}_j$ in the storage mapping is nothing but the vector of contraction moduli computed in this manner for the statement S_j at each level. The transformation matrix M_j is an $m \times n$ matrix constructed using the m storage hyperplanes found for the statement S_j as the m rows of the matrix. If a hyperplane $\Gamma_j^{(l)} = (\gamma_{l,1}, \gamma_{l,2}, \ldots, \gamma_{l,n})$, then the storage mapping matrix M_j is an $m \times n$ matrix with the l^{th} row $(\gamma_{l,1}\ \gamma_{l,2}\ \ldots\ \gamma_{l,n})$. The storage hyperplane offsets $\delta_j^{(1)}, \delta_j^{(2)}, \ldots, \delta_j^{(m)}$ make up the vector $\vec{\delta}_j$.

3.4.1 Global Conflict Set Specification

The global conflict set can be specified as a union of convex polyhedra, also called *conflict polyhedra*. A few of these polyhedra represent only the intra-statement conflicts, e.g. the conflict polyhedra in CS_0 and CS_1 (Fig. 2(c)). The remaining conflict polyhedra specify only the inter-statement conflicts with the convention that all conflicts represented in a given inter-statement conflict polyhedron involve indices from the same pair of local array spaces, for example, the conflict polyhedra in $CS_{0,1}$ and $CS_{1,0}$ (Fig. 2(d)). In essence, the domain and range of a conflict polyhedron must be sub-spaces of the same local array space or of two different ones.

Each integer point in a conflict polyhedron represents a particular conflict. The symmetricity of a conflict relation can be used to simplify the conflict set significantly. Hereafter, we assume 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. Furthermore, there may be multiple ways to specify a conflict set as a union of conflict polyhedra. We follow the convention that if the conflict relation $\vec{i} \bowtie \vec{j}$ is represented in the conflict set, the value for the conflicting index $\vec{j}$ must not be computed earlier, according to the given schedule, than that for the index $\vec{i}$. Finally, the conflict set specification must preferably be minimal in the sense that no two conflict polyhedra exist in the union such that their union is itself convex.

4. Finding Storage Hyperplanes

Storage hyperplanes only need to be found when the global conflict set is non-empty. Otherwise, all statements can write to a shared scalar variable. This section describes our approach for finding storage hyperplanes to exploit both intra-array and inter-array reuse.

4.1 Analyzing Intra-Statement Conflicts

In this section, we generalize the approach of Bhaskaracharya et al [4] for analyzing intra-statement conflicts defined over a global array space obtained by unifying the local array spaces of multiple statements. Suppose there are l conflict polyhedra $K_1, K_2, \ldots, K_l$ so that the global conflict set $CS = \cup_{i=1}^{i=l} K_i$. Consider a pair of conflicting indices $\vec{s}, \vec{t} \in A_j$ where A_j is the local array space of the statement S_j. In accordance with Definition 4, the hyperplane $\vec{\Gamma}_j$ with an offset δ_j, which needs to be found for statement S_j, will satisfy such an intra-statement conflict if $(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}) \neq 0$—the offset δ_j of the hyperplane is immaterial. This is also in accordance with the approach of Bhaskaracharya et al [4] and is akin to partitioning the local array space A_j to satisfy the intra-statement conflicts within it i.e., the intra-statement conflicts can be analyzed just as though we were dealing with a single statement despite all statements writing to a shared global array space.

Suppose $(\vec{u}_j.\vec{P} + w_j)$ is the upper bound on the conflict difference $(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t})$ for intra-statement conflicts associated with statement S_j. Similar to the constraint (1), we can formulate the following bounding constraint:

$$\begin{aligned} & (\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}) \leq (\vec{u}_j.\vec{P} + w_j) \leq (c\vec{P} + c) \\ \wedge \quad & -(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}) \leq (\vec{u}_j.\vec{P} + w_j) \leq (c\vec{P} + c). \end{aligned} \tag{3}$$

Additionally, following the rationale behind the constraint (2), a pair of binary decision variables x_{1i},x_{2i} for the intra-statement conflict polyhedron K_i can similarly be used to encode the satisfaction of such conflicts associated with statement S_j as follows:

$$\begin{aligned} & (\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}) \geq 1 - (1 - x_{1i})(c\vec{P} + c + 1) \\ \wedge \quad & (\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_J.\vec{t}) \leq -1 + (1 - x_{2i})(c\vec{P} + c + 1). \end{aligned} \tag{4}$$

In this way, each intra-statement conflict polyhedra is associated with its own pair of binary decision variables, both of which cannot simultaneously be equal to one. Suppose CS_{intra} represents the set of intra-statement conflict polyhedra. The number of intra-statement conflict polyhedra η_{intra}, all of whose conflicts are satisfied by the hyperplane found for the corresponding statement

can then be estimated as follows:

$$\eta_{intra} = \sum_{\forall i,\ K_i \in CS_{intra}} (x_{1i} + x_{2i}). \quad (5)$$

Bhaskaracharya et al [4] have shown that maximizing conflict satisfaction is a reasonably effective heuristic for exploiting intra-statement storage reuse opportunities. Essentially, fewer the number of conflicts which are left unsatisfied, fewer the number of storage hyperplanes required to satisfy all conflicts. Reasoning along similar lines, our primary objective is also to maximize the total number of intra-statement conflict polyhedra η_{intra} all of whose conflicts are satisfied. Such a greedy approach tries to minimize the number of storage hyperplanes required to satisfy intra-statement conflicts so that the dimensionality of the contracted local array spaces will be as small as possible.

4.2 Analyzing Inter-Statement Conflicts

Now that we have analyzed the intra-statement conflicts associated with a statement S_j, let us consider the inter-statement conflicts associated with it. Suppose S_k is another statement which writes to its local array space A_k and that $K_i \in CS$ is an inter-statement conflict polyhedron which specifies the inter-statement conflicts $\vec{s} \bowtie \vec{t}$ such that $\vec{s} \in A_j$ and $\vec{t} \in A_k$. In accordance with Definition 4, such a conflict is satisfied by the storage hyperplanes $\vec{\Gamma}_j$ and $\vec{\Gamma}_k$ with corresponding offsets δ_j and δ_k if $(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k) \neq 0$.

As described earlier, a finite upper bound of the form $(\vec{u}_j.\vec{P} + \vec{w}_j)$ can be enforced on the intra-statement conflict difference for S_j. Similarly, suppose that the statement S_j is associated with another $(\vec{u'}_j.\vec{P} + w'_j)$ which serves as the bound on any inter-statement conflict difference $(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k)$ associated with it. This leads to the following bounding constraints:

$$\begin{aligned} &(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k) \leq (\vec{u'}_j.\vec{P} + w'_j) \leq (c\vec{P} + c) \\ \wedge\ &-(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k) \leq (\vec{u'}_j.\vec{P} + w'_j) \leq (c\vec{P} + c). \end{aligned} \quad (6)$$

The inter-statement conflict difference could be positive, negative or equal to zero. Therefore, similar to the constraints in (4), the following constraints can be imposed on the inter-statement conflict difference $(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k)$ through a pair of decision variables x_{1i} and x_{2i} for the inter-statement conflict polyhedron K_i:

$$\begin{aligned} &(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k) \geq 1 - (1 - x_{1i})(c\vec{P} + c + 1) \\ \wedge\ &(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k) \leq -1 + (1 - x_{2i})(c\vec{P} + c + 1). \end{aligned} \quad (7)$$

The affine form of Farkas' lemma [6, 16] can be applied on the constraints formulated in (3), (6) and (4), (7), to obtain a set of linear equalities/inequalities by equating the coefficients of the loop variables, thereby eliminating them. Now, let CS_{inter} represents the set of inter-statement conflict polyhedra. The number of inter-statement conflict polyhedra η_{inter}, all of whose conflicts are satisfied by the hyperplanes found for the pair of statements associated with them can be estimated as follows:

$$\eta_{inter} = \sum_{\forall i,\ K_i \in CS_{inter}} (x_{1i} + x_{2i}). \quad (8)$$

4.3 A Greedy Objective

The resulting ILP system consists of constraints obtained due to the set of bounding constraints in (3) and (6), the decision constraints in (4) and (7) and also the constraints on η_{intra} and η_{inter} given by (5) and (8). Such constraints are derived for each of the l conflict polyhedra depending on whether they represent intra-statement conflicts or not.

As explained earlier, the primary objective of our greedy approach is to maximize intra-statement conflict satisfaction by maximizing η_{intra}. Another factor which needs to be considered while determining the storage hyperplanes is the storage size of the resulting modulo storage mapping for a statement S_j. In the successive modulo technique, the storage size of a modulo mapping obtained is computed as the product of the contraction moduli. The moduli themselves are computed along the canonical bases. The contraction modulus along a canonical basis is one plus the maximum conflict difference along it. Essentially, the canonical bases also serve as the storage hyperplane normals.

In general, the storage hyperplanes that we need to determine may not correspond to the canonical bases. Furthermore, when both intra-statement and inter-statement conflicts are considered together, the maximum conflict difference among the conflicts involving the statement S_j could be its maximum intra-statement conflict difference or its maximum inter-statement conflict difference, depending on which is greater. The former is bounded by

Algorithm 1 Find modulo storage mappings for r statements given a non-empty conflict set CS for the global array space A. $\vec{P}$ is the vector of program parameters.

procedure FIND-MODULO-MAPPINGS(A, CS, $\vec{P}$)
  $CS' \leftarrow CS$
  $m \leftarrow 1$
  while $CS' \neq \varnothing$ **do**
    $(\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}, e_0^{(m)}, e_1^{(m)}, \ldots, e_{r-1}^{(m)})$ $\leftarrow$ FIND-NEXT-HYPERPLANES(CS')
    Revise the conflict set CS' using (13) by revising the conflict polyhedra using (11) and (12)
    $m \leftarrow m + 1$
  for $0 \leq j \leq r-1$ **do**
    Let M_j be the transformation matrix for statement S_j constructed with hyperplanes $\Gamma_j^{(1)}, \Gamma_j^{(2)}, \ldots, \Gamma_j^{(m)}$ forming its rows
    Let $\vec{e_j} = \left(e_j^{(1)}, e_j^{(2)}, \ldots, e_j^{(m)}\right)$, the vector of contraction moduli
  return $(M_0, M_1, \ldots, M_{r-1}, \vec{e}_0, \vec{e}_1, \ldots, \vec{e}_{r-1})$
procedure FIND-NEXT-HYPERPLANES(CS')
  $C \leftarrow \varnothing$
  for all conflict polyhedra $K'_i \in CS'$ **do**
    Formulate bounding constraints using (3) and (6)
    Formulate satisfaction decision constraints using (4) and (7)
    Apply Farkas' lemma to each of the above constraints (formulated in steps 14 and 15) to obtain an equivalent set of linear equalities/inequalities and add them to C
    Add constraint (9) for trading off inter-statement conflict satisfaction if $K_i \in CS_{inter}$
  Add the constraint on η_{intra} and η_{inter} shown in (5) and (8) to C
  Compute lexicographic minimal solution as shown in (10) to obtain the hyperplanes $\Gamma_0, \Gamma_1, \ldots, \Gamma_{r-1}$ and the corresponding contraction modulo $e_0, e_1, \ldots, e_{r-1}$ for the r statements
  return $(\Gamma_0, \Gamma_1, \ldots, \Gamma_{r-1}, e_0, e_1, \ldots, e_{r-1})$

$(\vec{u}_j.\vec{P} + w_j)$ while the latter is bounded by $(\vec{u'}_j.\vec{P} + w'_j)$. Clearly, it is necessary to keep both of them to a minimum. We set $\vec{u'}_j$ to be element-wise greater than or equal to $\vec{u}_j$ and $w'_j \geq w_j$. We will see that this does not lead to a loss of optimization opportunity and is in fact used to prevent aggressive satisfaction of inter-statement conflicts which can lead to a large inter-statement conflict difference $(\vec{u'}_j.\vec{P} + w'_j)$. Consequently, since $(\vec{u'}_j.\vec{P} + w'_j)$ is greater than or equal to $(\vec{u}_j.\vec{P} + w_j)$, as our secondary objective, we try to minimize the contraction modulus for each statement by first minimizing the bound $(\vec{u}_j.\vec{P} + w_j)$ associated with it. Even if $(\vec{u'}_j.\vec{P} + w'_j)$ proves to be a loose bound on the inter-statement conflict difference, giving precedence to the minimization of $(\vec{u}_j.\vec{P} + w_j)$ over that of $(\vec{u'}_j.\vec{P} + w'_j)$ does not affect the final contraction modulus.

Note that it is possible to satisfy all inter-statement conflicts in one go by choosing the canonical basis for the outermost dimension in the global array space as the first storage hyperplane for every statement. However, premature satisfaction of inter-statement conflicts in this way can destroy any opportunity available for inter-statement reuse. This is why the primary and secondary objectives are focused mainly on satisfying intra-statement conflicts. It is equivalent to solving the problem of intra-array reuse for each statement separately. This is in line with the general approach of Lefebvre and Feautrier [12], who also give precedence to exploiting intra-statement storage reuse over inter-statement storage reuse. However, while maximizing intra-statement conflict satisfaction, it is also possible to satisfy inter-statement conflicts. A particularly interesting case is when all the inter-statement conflicts are satisfied as a side-effect of satisfying intra-statement conflicts. In other words, if no hyperplane is needed to exclusively satisfy the inter-statement conflicts, it means that inter-statement storage reuse has already been achieved. However, if inter-statement conflicts are satisfied too aggressively, this may have to be at the expense of increasing the inter-statement conflict difference too much, leading to a much higher contraction modulus. Specifically, if $(\vec{u'}_j.\vec{P} + w'_j)$ will exceed $(\vec{u}_j.\vec{P} + w_j)$ by more than a constant additive factor, we choose to leave the inter-statement conflicts of S_j unsatisfied. Inter-statement conflict satisfaction is thus traded off in favor of a smaller contraction modulus for the statement S_j. If $\vec{u}_j = (u^{(0)}, u^{(1)}, \ldots, u^{(\rho-1)})$ and $\vec{u}'_j = (u'^{(0)}, u'^{(1)}, \ldots, u'^{(\rho-1)})$, ρ being the number of parameters involved, such a trade-off can be specified by the following constraint for each inter-statement conflict polyhedron K_i:

$$0 \leq \sum_{p=0}^{\rho-1} \left(u'^{(p)} - u^{(p)} \right) \leq (1 - x_{1i} - x_{2i})(c\rho). \qquad (9)$$

This constraint ensures that the inter-statement conflict polyhedron associated with statement S_j is allowed to be satisfied only if $u'^{(p)} = u^{(p)}$ for $p = 0, 1, \ldots, \rho - 1$. With these additional constraints for every statement S_j added to the ILP system, we can proceed to maximize inter-statement conflict satisfaction as well by first maximizing η_{inter} while minimizing the bound $(\vec{u'}_j.\vec{P} + w'_j)$ on the inter-statement conflict difference of every statement S_j.

η_{intra} and η_{inter} are at most equal to $|CS_{intra}|$ and $|CS_{inter}|$ respectively. If $\eta'_{intra} = (|CS_{intra}| - \eta_{intra})$ and $\eta'_{inter} = (|CS_{inter}| - \eta_{inter})$, the fourfold objective of maximizing η_{intra}, minimizing $(\vec{u}_j.\vec{P} + w_j)$, maximizing η_{inter} and finally, minimizing $(\vec{u'}_j.\vec{P} + w'_j)$ for each statement S_j can be achieved simultaneously by finding a lexicographically minimal solution as follows:

$$\begin{aligned} minimize_{\prec} \{ & \eta'_{intra}, u_0^{(0)}, u_0^{(1)}, \ldots, u_0^{(\rho-1)}, w_0, \ldots \\ & \ldots, u_{r-1}^{(0)}, u_{r-1}^{(1)}, \ldots, u_{r-1}^{(\rho-1)}, w_{r-1}, \\ & \eta'_{inter}, u'^{(0)}_0, u'^{(1)}_0, \ldots, u'^{(\rho-1)}_0, w'_0, \ldots \\ & \ldots, u'^{(0)}_{r-1}, u'^{(1)}_{r-1}, \ldots, u'^{(\rho-1)}_{r-1}, w'_{r-1} \}. \end{aligned} \qquad (10)$$

Therefore, if no inter-statement conflict polyhedron associated with the statement S_j is satisfied, the contraction modulus is taken to be equal to $(\vec{u}_j.\vec{P} + w_j + 1)$. Otherwise, it equals $(\vec{u'}_j.\vec{P} + w'_j + 1)$.

4.4 Finding Storage Hyperplanes Iteratively

All the conflicts in the global conflict set may not necessarily be satisfied by the first set of storage hyperplanes found for every statement. It is therefore necessary to eliminate the conflicts, which have been satisfied so far, from the conflict set. Such a revised conflict set can then be used to find another set of storage hyperplanes which can satisfy a few or all of the remaining conflicts.

Suppose the hyperplanes $\vec{\Gamma}_0, \vec{\Gamma}_1, \ldots, \vec{\Gamma}_{r-1}$ have been found for the statements $S_0, S_1, \ldots, S_{r-1}$ respectively, based on the global conflict set $CS = K_1 \cup K_2 \cup \cdots \cup K_l$. Furthermore, let e_j be the contraction modulus determined for statement S_j. Consider a conflict polyhedra K_i which specifies conflicts between $\vec{s} \bowtie \vec{t}$. If K_i is an intra-statement conflict polyhedron, the conflicts in it which are not satisfied by the storage hyperplane $\vec{\Gamma}_j$ satisfy the constraint $(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t} = 0)$. This means that an intra-statement conflict polyhedron can be revised by adding the constraint $(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t} = 0)$ to eliminate from it the conflicts which have been satisfied:

$$\forall K_i \in CS_{intra}, \quad K'_i = K_i \cap \{(\vec{s}, \vec{t}) \mid \vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t} = 0\}. \qquad (11)$$

On the other hand, suppose K_i is an inter-statement conflict polyhedron representing conflicting indices from the local array spaces A_j and A_k. In this case, a few unsatisfied conflicts may have conflict difference $(\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k)$ equal to 0. Additionally, even a few inter-statement conflicts whose conflict difference is not zero have to be treated as unsatisfied. This is due to the trade-off involved in inter-statement conflict satisfaction. It can be seen that such conflicts satisfy the constraint $\mid \vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k \mid \geq min(e_j, e_k)$ i.e., if the conflict difference exceeds or equals the contraction modulus computed either for the statement S_j or that for S_k, it must be treated as unsatisfied. An inter-statement conflict polyhedron can therefore be revised as follows:

$$\begin{aligned} \forall K_i \in CS_{inter}, K'_i = K_i \cap \{(\vec{s}, \vec{t}) \mid \\ (\vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k = 0) \\ \vee \mid \vec{\Gamma}_j.\vec{s} + \delta_j - \vec{\Gamma}_k.\vec{t} - \delta_k \mid \geq min(e_j, e_k)\}. \end{aligned} \qquad (12)$$

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

$$CS' = \cup_{1 \leq i \leq l} K'_i. \qquad (13)$$

The next set of storage hyperplanes can now be found using the revised conflict set CS' instead of the original conflict set CS. Note that if all the conflicts in a conflict polyhedron K_i are satisfied by the hyperplane $\vec{\Gamma_j}$ (and the hyperplane $\vec{\Gamma_k}$, if it is an inter-statement conflict polyhedron), none of these conflicts will be present in the revised conflict set CS' as all of them are eliminated due to the addition of the above constraints. In this way, the global array space is successively partitioned for each statement until all conflicts are satisfied i.e., until conflict sets are eventually revised to empty sets. At each step, the contraction moduli are also computed for every statement.

Algorithm 1 summarizes the partitioning-based approach to find modulo storage mappings for r statements $S_0, S_1, \ldots, S_{r-1}$ given the global conflict set CS. The main procedure, FIND-MODULO-MAPPINGS (line 1), determines the m storage hyperplanes iteratively for each statement, revising the conflict set at each step as described above (lines 4-6). The procedure, FIND-NEXT-HYPERPLANES (line 11), sets up the ILP system (lines 13-17) necessary to determine the required storage hyperplanes (line 19) and the corresponding contraction moduli.

4.5 Array Decoalescing

Our partitioning approach is based on satisfying conflicts in the global array space A. Consequently, the m-dimensional modulo storage mapping obtained for each statement S_j is also valid for this global array space. It is of the form $A[\vec{i}] \rightarrow A[M_j\vec{i} \bmod \vec{e}_j]$ where M_j is the storage mapping matrix with the m hyperplanes found for S_j serving as its rows. The vector $\vec{e}_j$ is the vector of m corresponding contraction moduli $(e_j^{(1)}, e_j^{(2)}, \ldots, e_j^{(m)})$. In effect, these storage mappings map all statements to a shared global array space.

The graph coloring approach by Lefebvre and Feautrier [12] tries to lump together contracted arrays of different statements into a shared data structure by computing their rectangular hull. Coalescing the contracted array spaces into a rectangular hull in this manner can sometimes lead to excessive storage when compared to leaving them uncoalesced. For example, coalescing a $2 \times N$ and an $N \times 2$ array can increase the overall storage requirement dramatically to N^2. Since our heuristic attempts to find storage mappings for each statement based on an already shared global array space, such a scenario is possible even with our approach. It is therefore necessary to decoalesce such arrays so that the corresponding statements can write to their own separate array spaces.

Consider two statements S_j and S_k. If the contraction modulus vector $\vec{e}_j$ is element-wise greater than or equal to the vector $\vec{e}_k$, or vice versa, the two statements can clearly write to the same array. In other words, the contracted array space for one can be completely embedded inside the other. If this condition does not hold, it is better to map the two statements to arrays of different names in order to avoid the storage overhead incurred as a side-effect of computing the rectangular hull. The condition specified above for coalescing is for a complete fit of one contracted array within another. But it can often be relaxed slightly so that array coalescing is allowed so long as the contraction moduli of the two array space involved are of comparable sizes. Since the contraction moduli are often parametric, the relative sizes of the i^{th} contraction moduli $e_j^{(i)}$ and $e_k^{(i)}$ (in the vectors $\vec{e}_j$ and $\vec{e}_j$) can be estimated by considering the contribution of their parametric parts alone. This can be done by adding up the coefficients of the parameters in $e_j^{(i)}$ and $e_k^{(i)}$ respectively. Let $\Delta(e_j^{(i)})$ be the sum of the parametric coefficients in the contraction modulus $e_j^{(i)}$. Then the condition for leaving the contracted array spaces of two statements coalesced can be re-stated as $(\Delta(e_j^{(1)}), \Delta(e_j^{(2)}), \ldots, \Delta(e_j^{(m)})$ being element-wise greater than or equal to $(\Delta(e_k^{(1)}), \Delta(e_k^{(2)}), \ldots, \Delta(e_k^{(m)})$ (or vice versa).

An undirected array coalescing graph G with r nodes can then be constructed such that each node in the graph corresponds to a given statement. If a pair of statements S_j and S_k can write to the same array in accordance with the condition for array coalescing, the graph G has an edge between the nodes corresponding to the two statements. All statements belonging to the same connected component in the graph can then be mapped to the same array. For example, consider the case of whether a $2 \times N$ storage mapping should be coalesced with a $N \times 2$ one. The contraction modulus vectors are nothing but the vector of the array sizes—$(2, N)$ and $(N, 2)$ respectively. The parametric coefficient sums for them are therefore $(0, 1)$ and $(1, 0)$. Since neither of these two vectors is element-wise less than the other, the two arrays are better left decoalesced. Clearly, if we construct an undirected graph as described above for these two, their corresponding nodes would be in separate connected components. Now, suppose, there is another statement which requires a contracted array space of size (N, N). This third array can be coalesced with both of the other two arrays. Consequently, the array coalescing graph will have only one connected component due to which all the arrays will be coalesced together.

Decoalescing the array spaces as described above divides the given set of statements into equivalence classes. Statements in the same equivalence class can write to an array of the same name using the storage mapping obtained for each of them. The sizes of the m-dimensions for such an array are computed as the maximum of the corresponding contraction moduli computed for each statement. Since our heuristic has a fairly fast running time, it can be run again on each of these equivalence classes of statements, thereby completely ignoring conflicts across statements in different equivalence classes.

Example revisited Consider again the example in Fig. 2. In Fig.3, the intra-statement conflicts are shown in red whereas the inter-statement conflicts are shown in orange. Now, suppose the storage hyperplanes found for both the statements S_0 and S_1 happen to be the same one—the canonical hyperplane $(0, 1, 0)$ with a zero offset. In such a scenario, only the inter-statement conflicts shown in green would be satisfied. On the other hand, if the zero offset canonical hyperplane $(0, 0, 1)$ is considered for both the statements instead of $(0, 1, 0)$, it would satisfy all but the inter-statement conflicts represented in $CS_{1,0}$ with a maximum conflict difference of N. However, even these conflicts can be satisfied by modifying the hyperplane choice to $(0, -1, 1)$, again with a zero offset, while increasing the maximum inter-statement conflict difference to $(N + 1)$. Note that the intra-statement conflict differences do not change. Several other hyperplanes such as $(0, -2, 1)$, $(0, -3, 1)$, which can also satisfy all conflicts, are ignored as they would result in a bigger contraction modulus for both the statements. Furthermore, since all conflicts are satisfied by the hyperplane $(0, -1, 1)$ itself, there is no need to find any more partitioning hyperplanes. Moreover, the storage hyperplane and the contraction modulus found for the two statements happen to be the same. Consequently, the resulting storage mapping for them is also the same: $A[j, t, i] \rightarrow A[(i - t) \bmod (N + 1)]$ for $j = 0, 1$. The same array of size $(N + 1)$ can be written to by both the statements, thereby ensuring inter-statement storage reuse. This mapping provides a storage size requirement which is better than that obtained using the successive modulo technique by a factor 2. In fact, the modulo storage mapping is storage optimal.

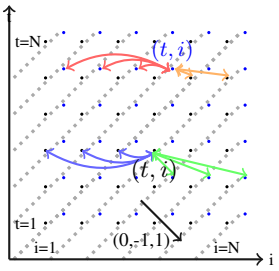


Figure 5. Storage hyperplane $(0, -1, 1)$ satisfies all conflicts.

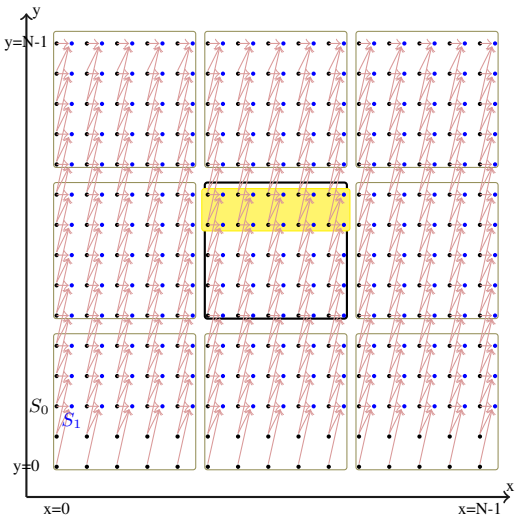
This example also shows that exploiting storage reuse can sometimes expose copy elimination opportunities. The statement S_1, after storage optimization, gets rewritten as $A[(i - t) \bmod (N + 1)] = A[(i - t) \bmod (N + 1)]$ which is a redundant copy operation and so, can be eliminated. Consequently, the loop enclosing statement S_1 can also be eliminated. The resulting code is a perfect loop-nest with statement S_0.

```
// 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

```
#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)
/*S0*/    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)
/*S1*/      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 the arrays blurx and out,
// the write accesses become 4-d accesses
// #define blurx(x,y) ((y>=0)?A0[ty,tx,x,y]
//                    :A0[ty-1,tx,x,B+y])
// #define out(x,y) A1[ty,tx,x,y]
```

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



(c) Tiled execution of blur filter. Live-out data for the tile in yellow. The black and blue dots represent instances of the statements S_0 and S_1 respectively.

$$CS_0 = ((x = x') \wedge (y' > y) \wedge (x, y), (x', y') \in A_{0T})$$
$$\vee((x' > x) \wedge (y \leq B-1) \wedge (y \geq B-2) \wedge (x, y), (x', y') \in A_{0T})$$
$$CS_1 = ((x' > x) \wedge (x, y), (x', y') \in A_{1T})$$
$$\vee((x = x') \wedge (y' > y) \wedge (x, y), (x', y') \in A_{1T})$$
$$CS_{0,1} = ((x = x') \wedge (y' - 1 \leq y) \wedge (x, y) \in A_{0T}, (x', y') \in A_{1T})$$
$$\vee((x' \geq x) \wedge (y \leq B-1) \wedge (y \geq B-2)$$
$$\wedge (x, y) \in A_{0T}, (x', y') \in A_{1T})$$
$$CS_{1,0} = ((x' > x) \wedge (x, y) \in A_{1T}, (x', y') \in A_{0T})$$

(d) The geometrical representations of these conflict sets are shown on the right in corresponding order.

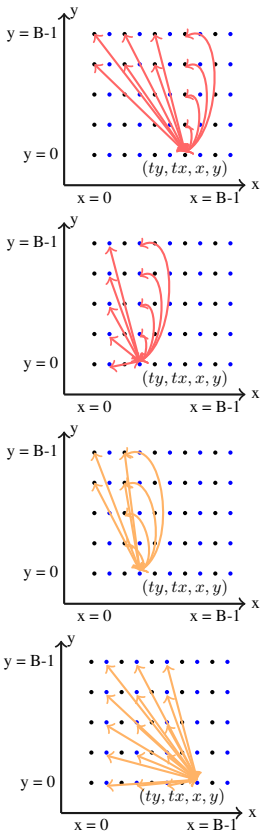


Figure 6. A geometrical representation of the tiled execution of blur filter is show in Fig.6(c); the conflict sets representing the intra-tile conflicts $(j, x, y) \bowtie (j', x', y')$ in the global array space A are shown in Fig.6(c); statements S_0 and S_1 write to the data tiles A_{0T} and A_{1T} respectively

Correctness and Termination The objective of successively partitioning the global array space is to ultimately satisfy all conflicts. As observed by Bhaskaracharya et al [4], a storage hyperplane that is linearly dependent on the hyperplanes found in earlier iterations cannot satisfy any new intra-statement conflicts. Furthermore, if there are some inter-statement conflicts associated with a statement that are are still not satisfied after all the associated intra-statement conflicts have been satisfied, exactly one more storage hyperplane needs to be found for the statement in order to satisfy them. Therefore, the iterative process is guaranteed to terminate.

5. Examples

This section discusses storage mappings obtained by our technique on a few examples drawn from real-world scenarios to help understand it better.

5.1 Blur filter

Although stencils frequently occur in scientific applications as time-iterated computations such as the example shown in Fig. 1, in image processing pipelines, the stencil computation may not necessarily be time-iterated. Instead, a pipeline stage may apply a particular stencil once, before propagating the output to the next stage, which in turn 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-Kelly et al [15] for their work on the Halide DSL compiler. Consider the loop-nest of a 2-stage blur (in Fig. 6(a)). The first stage performs a horizontal blur of the input image. The second stage then performs a vertical blur of the horizontal blur output to produce the final isotropic blur. The poor producer-consumer data locality can be improved through tiling.

A tiled version of the blur filter code is shown in Fig.6(c). The schedules for the statements S_0 and S_1 can be expressed as $\theta(S_0(ty, tx, x, y)) = (ty, tx, x, 0, y)$ and $\theta(S_1(ty, tx, x, y)) = (ty, tx, x, 1, y)$. The column-wise processing is interleaved to improve locality so that a column of $blurx$ within the tile, once computed, is immediately read for the vertical blur along the same column. The top two rows of each data tile of $blurx$ constitute its live-out data for such a schedule (refer 6(c)). When a total expansion of the array spaces written to by the statements S_0 and S_1 is performed, their write accesses become 4-dimensional accesses on their local array spaces A_0 and A_1 respectively, which have the same size and shape as their iteration domains (refer Fig.6(b)). Suppose A is the global unified array space such that $A[j] = A_j$ for $j = 0, 1$. For brevity's sake, we only consider the problem of optimizing the storage for a particular compute tile (ty, tx) which writes to the unified data tile $A_T = A[ty, tx]$. Let $A_{jT} = A_T[j]$ represent the data tile written by the statement S_j.

```
#define isbound(i,j) (i==0)||(i==(N-1))
                     ||(j==0)||(j==(N-1))
for(int i=0; i<N; ++i)
  for(int j=0; j<N: ++j)
/*S0*/ A0[i][j]=(!isbound(i,j)) ? a[i][j]
                +(a[i-1][j]+a[i+1][j]+a[i][j-1]
                +a[i][j+1]) : a[i][j];
for(int i=0; i<N; ++i)
  for(int j=0; j<N: ++j)
/*S1*/ A1[i][j]=(!isbound(i,j)) ? A0[i][j]
                +(A0[i-1][j]+A0[i+1][j]+A0[i][j-1]
                +A0[i][j+1]) : A0[i][j];
for(int i=0; i<N; ++i)
  for(int j=0; j<N: ++j)
/*S2*/ A2[i][j]=(!isbound(i,j)) ? A1[i][j]
                +(A1[i-1][j]+A1[i+1][j]+A1[i][j-1]
                +A1[i][j+1]) : A1[i][j];
for(int i=0; i<N; ++i)
  for(int j=0; j<N: ++j)
/*S3*/ A3[i][j]=(!isbound(i,j)) ? A2[i][j]
                +(A2[i-1][j]+A2[i+1][j]+A2[i][j-1]
                +A2[i][j+1]) : A2[i][j];
for(int i=0; i<N; ++i)
  for(int j=0; j<N: ++j)
/*S4*/ A4[i][j]=(!isbound(i,j))? A3[i][j]
                +(A3[i-1][j]+A3[i+1][j]+A3[i][j-1]
                +A3[i][j+1]) : A3[i][j];
```

(a) Smoothing using the Jacobi stencil.

$$CS_k = \{(k,i,j) \bowtie (k,i',j') \mid (k,i,j),(k,i',j') \in A \wedge ((i<i') \vee ((i=i') \wedge (j<j')))\}$$

(b) The intra-statement conflict set specification for statement S_k for $k = 0, 1, \ldots, 4$.

$$CS_{k,k+1} = \{(k,i,j) \bowtie (k+1,i',j') \mid (k,i,j),(k+1,i',j') \in A \wedge (i \geq i') \vee ((i+1=i') \wedge (j>j'))\}$$

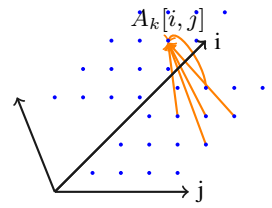
(c) The inter-statement conflict set specification for statements S_k and S_{k+1} for $k = 0, 1, \ldots, 3$.

$$CS = (\vee_{k=0,1,\ldots,4} CS_k) \vee (\vee_{k=0,1,\ldots,3} CS_{k,k+1})$$

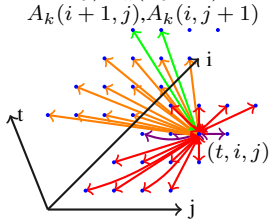
(d) The global conflict set specification.

$$\begin{aligned}
S_0 &: A[0,i,j] \rightarrow A[(i+3) \mod (N+2)][j \bmod N] \\
S_1 &: A[1,i,j] \rightarrow A[(i+1) \mod (N+2)][j \bmod N] \\
S_2 &: A[2,i,j] \rightarrow A[(i-1) \mod (N+2)][j \bmod N] \\
S_3 &: A[3,i,j] \rightarrow A[(i-3) \mod (N+2)][j \bmod N] \\
S_4 &: A[4,i,j] \rightarrow A[(i-5) \mod (N+2)][j \bmod N]
\end{aligned}$$

(e) The storage mapping obtained using our heuristic.



(f) Inter-statement flow dependences: $A_{k+1}(i,j)$ depends on $A_k(i,j)$,$A_k(i-1,j)$,$A_k(i,j-1)$, $A_k(i+1,j)$,$A_k(i,j+1)$



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

Figure 7. Storage optimization of Jacobi 2-d smoothing in multi-grid methods

The intra-tile conflict sets representing the conflicts $(j,x,y) \bowtie (j',x',y')$ are specified in Fig.6(d). CS_0 and CS_1 represent the intra-statement conflict sets for the statements S_0 and S_1 respectively. $CS_{0,1}$ represent the inter-statement conflicts which ensure that a value computed by the statement S_1 does not overwrite a value computed by statement S_0 before its last use. Similarly, the inter-statement conflict set $CS_{1,0}$ avoids a premature over-write by statement S_0 of a value computed by statement S_1. Note that the storage mappings obtained using the successive modulo technique $A_j[ty,tx,x,y] \rightarrow A_j[ty,tx,x \bmod B, y \bmod B]$ do not contract storage at all. Also, the contracted arrays for A_0 and A_1 cannot be fused into one using the rectangular hull method.

All the intra-statement conflicts of S_0 in CS_0 can be satisfied at once by the hyperplane $(0,2,-1)$ with a maximum intra-statement conflict difference of $(3B-3)$. However, since all values computed by S_1 are live-out, it is can be seen that not all of the intra-statement conflicts of S_1 can be satisfied immediately. Instead, a hyperplane such as $(0,1,0)$ can be used to satisfy one of the two intra-statement conflict polyhedra in CS_1 with a maximum intra-statement conflict difference of $(B-1)$. Furthermore, this choice of hyperplanes leads to a maximum inter-statement conflict difference of $2B-1$. Since this exceeds $(B-1)$ by more than an additive constant factor, our heuristic does not treat any of the inter-statement conflict polyhedra as satisfied. On the second iteration, after revising the conflict polyhedra, since no intra-statement conflicts of S_0 need to be satisfied, the hyperplane found for it is $(0,0,0)$ with 0 serving as the maximum intra-statement conflict difference. The hyperplane found for S_1 is $(0,0,1)$ with its maximum intra-statement conflict difference being $(B-1)$. Again, no inter-statement conflict polyhedra can be satisfied as the maximum inter-statement conflict difference turns out to be $(B-1)$ which is greater than the maximum intra-statement conflict difference of S_0. Finally, on the third iteration, the hyperplane $(0,0,0)$ chosen for both S_0 and S_1 but with corresponding offsets 0 and 1 satisfy all the remaining inter-statement conflicts—the resulting contraction modulus is 2. The resulting intra-tile mappings are $S_0 : A_T[0,x,y] \rightarrow A_T[(2x-y) \bmod (3B-2), 0 \bmod 1, 1 \bmod 2]$ and $S_1 : A_T[1,x,y] \rightarrow A_T[x \bmod B, y \bmod N, 0 \bmod 2]$. The contracted global array space can be decoalesced because S_0 clearly needs a smaller dimensional array space which does not fit into the array space of S_1. After decoalescing so that S_0 and S_1 write to arrays A'_0 and A'_1 respectively and eliminating the constant accesses, the final mappings obtained are $S_0 : A_0[ty,tx,x,y] \rightarrow A'_0[ty,tx,(2x-y) \bmod (3B-2)]$ and $S_1 : A_1[ty,tx,x,y] \rightarrow A'_1[ty,tx,x \bmod B, y \bmod N]$.

5.2 Smoothing

The geometric multigrid algorithm [7] for solving partial differential equations consists of different stages such as smoothing, interpolation, and restriction. All of these can be specified as stencil computations. The smoothing stage consists of repeated applications of a stencil operation on the given grid. A high-level specification of the computation can be provided through a DSL such as PolyMage [13]. Fig. 7(a) shows a 5-step smoothing stage implemented using the Jacobi method. Each statement S_k writes to its local array space A_k which has the same size and shape as the iteration domain of S_k. The flow dependences are as as shown Fig.7(f)). The last use of a value computed by the statement instance $S_{k-1}(i,j)$ is in $S_k(i+1,j)$. Now suppose that all the local array spaces are unified into a global array space A on which all the statements operate. The intra-statement conflict set CS_k for the statement S_k can then be specified as shown in Fig. 7(b)—the index (k,i,j) in the global array space conflicts with indices of all values computed later by the statement S_k. Furthermore, the inter-statement conflict set $CS_{k,k+1}$, shown in Fig. 7(c), specifies that the index (k,i,j) conflicts with the all indices lexicographically less than $(k+1,i+1,j)$ (since $S_{k+1}(k+1,i+1,j)$ is when $A(k,i,j)$ is last used). A geometric representation of the conflict sets CS_k and $CS_{k,k+1}$ is shown in Fig. 7(g). The former consists of the conflicts shown in red and violet, whereas the conflicts in orange and green represent the inter-statement conflicts.

Suppose the successive modulo technique is applied individually for each local array space separately. Since there is no scope for intra-statement storage reuse, none of them can then be contracted

Benchmark	Modulo storage mapping		Reduction (approx.)	SMO time
	baseline	SMO		
1-d stencil (Fig.2)	$S_0 : A_0[t \bmod 1, i \bmod N]$ $S_1 : A_1[t \bmod 1, i \bmod N]$	$A[(i-t) \bmod (N+1)]$ $A[(i-t) \bmod (N+1)]$	2	0.055s
2-d stencil	$S_0 : A_0[t \bmod 1, i \bmod N, j \bmod N]$ $S_1 : A_1[t \bmod 1, i \bmod N, j \bmod N]$	$A[(i-3t+1) \bmod (N+2), j \bmod N]$ $A[(i-3t) \bmod (N+2), j \bmod N]$	2	0.633s
3-d stencil	$S_0 : A_0[t \bmod 1, i \bmod N, j \bmod N, k \bmod N]$ $S_1 : A_1[t \bmod 1, i \bmod N, j \bmod N, k \bmod N]$	$A[(i-3t) \bmod (N+2), j \bmod N, k \bmod N]$ $A[(i-3t-1) \bmod (N+2), j \bmod N, k \bmod N]$	2	22.57s
jacobi-2d-smoothing (Fig.7)	$S_k : A_{k\%2}[i \bmod N, j \bmod N]$	$A[(i+3-2k) \bmod (N+2), j \bmod N]$	2	4.846s
blur-tiled (Fig.6)	$S_0 : A_0[ty, tx, x \bmod B, y \bmod B]$ $S_1 : A_1[ty, tx, x \bmod B, y \bmod B]$	$A'_0[ty, tx, (y-2x) \bmod (3B-2)]$ $A'_1[ty, tx, x \bmod B, y \bmod B]$	$\frac{B}{3}$ 1	0.738s
unsharp-tiled	$S_0 : A_0[z, ty, tx, x \bmod B, y \bmod B]$ $S_1 : A_1[z, ty, tx, x \bmod B, y \bmod B]$	$A'_0[z, ty, tx, (y-4x) \bmod (5B-4)]$ $A'_1[z, ty, tx, -y \bmod B, x \bmod B]$	$\frac{B}{5}$ 1	1.013s

Table 1. Storage reduction obtained using our approach (SMO) compared to the baseline (successive modulo [12] followed by rectangular hull), where B is the loop blocking factor

further. Moreover, the resulting mapping $A_k[i,j] \rightarrow A_k[i,j]$ for the statement S_k implies that S_k and S_{k+1} cannot share the rectangular hull of A_k and A_{k+1} as the common data structure due to the inter-statement conflict $(k,i,j) \bowtie (k+1,i,j)$ among other ones. Consequently, a graph coloring on the array interference graph which treats each array A_k as interfering with A_{k+1} would lead to a storage mapping $A_k[i,j] \rightarrow A_{k\%2}[i,j]$ i.e., the statements alternate between two arrays. The total storage requirement would then be $2N^2$.

Applying our heuristic on the global conflict set CS (shown in Fig. 7(d)) it can be seen that no storage hyperplane can satisfy all the intra-statement conflicts at once. The hyperplane $(0,1,0)$ satisfies the intra-statement conflicts in red with $(N-1)$ being the maximum intra-statement conflict difference. Consequently, our heuristic explores the space of alternative hyperplanes that not only satisfy the red conflicts but can also satisfy some inter-statement conflicts with the maximum inter-statement conflict difference exceeding the maximum intra-statement conflict difference by at most a constant additive factor. Indeed, the storage hyperplanes $(0,1,0), (1,1,0), (0,1,0)$, $(-1,1,0)$ and $(-1,1,0)$ for the statements S_0, S_1, S_2, S_3, S_4 respectively, with corresponding offsets $3, 0, -1, 0$ and -1, can together satisfy the orange and green conflicts as well as the satisfying the red ones on their own. They do so with a maximum inter-statement conflict difference of $(N+1)$. The remaining conflicts in violet can then be easily satisfied by choosing the hyperplane $(0,0,-1)$ for all the statements with a zero offset. The resulting storage mapping is as shown in Fig 7(e). Array decoalescing does not map the statements to different arrays as all of them write to a common $(N+2) \times N$ array. Note that the resulting storage requirement of (N^2+2N) is only marginally greater than the optimal storage requirement of (N^2+N) here, which is the maximum number of live values across all points during execution.

6. Implementation and Practical Impact

We have implemented an automatic storage optimizer, SMO, based on our heuristic. It uses ISL [20] (version isl 0.12.2) with GLPK (GNU Linear Programming kit) [8] version 4.45 as the ILP solver. The input to SMO in a global conflict set specification consisting of both inter-statement and intra-statement conflict polyhedra. The output obtained is the modulo storage mapping using our technique for each statement. Table 1 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. The suite of benchmarks includes time-iterated $1-$d, $2-$d and $3-$d stencils (implemented with ping-pong fashion as shown in Figure 1), tiled versions of blur filter and unsharp mask image processing kernels [13], and Jacobi smoothing iterations used in Multigrid methods [7]. The overall storage was approximately reduced by a factor of 2 for all benchmarks—due to the increase in intra-statement reuse for the unsharp-tiled and blur-tiled benchmarks and due to improved inter-statement reuse for the rest.

Benchmark	Problem size	Execution time		Speedup
		baseline	SMO	
1-d-stencil-ping-pong	N= 524288, T=256	0.411s	0.388s	1.059
2-d-stencil-ping-pong	N= 16384, T=16	39.65s	33.84s	1.172
2-d-stencil-ping-pong	N= 32768, T=8	85.07s	69.27s	1.228
3-d-stencil-ping-pong	N=128, T=512	22.70s	22.96s	0.988
3-d-stencil-ping-pong	N=256, T=32	11.17s	12.11s	0.922
3-d-stencil-ping-pong	N=512, T=32	88.71s	114.0s	0.778
jacobi-2d-smoothing	N=4096, 3 steps	2.455s	2.247s	1.092
jacobi-2d-smoothing	N=4096, 5 steps	2.896s	2.706s	1.070
jacobi-2d-smoothing	N=4096, 9 steps	3.820s	3.758s	1.016
unsharp-tiled	N=4096, B=256	1.337s	0.679s	1.969
blur-tiled	N=8192, T=512	0.046s	0.044s	1.045

Table 2. Benchmark performance with the storage mappings of Table 1

Finding storage hyperplanes using our technique relies on integer linear programming; in addition, Fourier-Motzkin variable elimination is used to eliminate Farkas multipliers. Despite their worst case exponential complexities, the compile time measurements in Table 1 (SMO time) indicate that they are usually quite fast in practice in our context. The relatively long time required to analyze the 3-d stencil benchmark is primarily due to the comparatively higher dimensionality of its global array space (5 dimensions) together with a total of 12 conflict polyhedra which need to be analyzed for it.

Although our primary concern in this work has been to optimize storage, we also examined its performance implications. Table 2 compares execution time of the benchmarks optimized for storage. The baseline version was optimized using the successive modulo technique followed by the rectangular hull method. For the benchmarks blur-tiled and unsharp-tiled, the tiles were executed in parallel using OpenMP. The benchmarks were compiled with GCC (version 4.8.1) with flags "-O3 -fopenmp" and run on all cores of an Intel Xeon E5-2680 v2 dual-socket machine with 8 cores per socket and a total of 64 GB of DDR3 1600 MHz RAM. The performance numbers presented in Table 2 are medians of five trial runs. As can be seen, there are improvements ranging from 5.9% to 96.9% for the selected benchmarks. Except for the 3-d stencil, we did not notice any significant slowdown in performance. We believe this slowdown is due to the more complex modulo mapping,

the overhead from which is relatively higher in the case of a 3-d stencil due to a higher number of array accesses.

7. Related Work

Most storage optimization techniques in the literature are intra-array ones. This includes those of Wilde and Rajopadhye [21], Lefebvre and Feautrier [12], Strout et al. [17], Quilleré and Rajopadhye [14], Thies et al. [18, 19], Darte et al. [3, 5], and Bhaskaracharya et al. [4]. Among these, only [12] proposes an inter-array reuse technique that can be used in conjunction with other intra-array techniques in a decoupled and an orthogonal way. On the other hand, our approach here presents the first unified intra-array and inter-array optimization technique. The example in Figure 1 presented motivation for such a unified approach.

Bhaskaracharya et al. [4] introduced the notion of storage hyperplanes and model the intra-array storage optimization as one of finding the right orientation for the storage hyperplanes. They claimed significant improvements in the quality of intra-array storage optimization over previous techniques. Our work here builds on [4], generalizing it to a global array space allowing both intra and inter array storage optimization, and encoding objective functions for both.

The inter-array compaction heuristic presented by Lefebvre and Feautrier [12] is decoupled from intra-array compaction; it thus misses mappings that can be obtained by taking a holistic view of all conflicts. Further discussion was provided in Section 2. De Greef et al. [9, 10]'s approach works by looking at a linearization of the array space, and then computing the maximum of the address differences between memory cells that are simultaneously live at any execution point. Such an approach misses contraction along non-canonical directions. Furthermore, the compatibility and mergeability checks for the reuse of contracted arrays are not capable of exploiting inter-array reuse opportunities such as the one in Fig 1.

8. Conclusion

Automatic solutions to the storage optimization problem are crucial for high-level and domain-specific language compilers, where a code generation scheme unaware of array reuse leads to excessive storage. For scaling to large data sets and for performance, it is necessary to reduce the memory footprint. A decoupled approach for intra and inter-array optimization, in spite of being powerful in its own class, is only capable of local decisions on compressing storage. We addressed this problem by proposing a single unified solution to perform intra and inter-array memory optimization. Experimental results show significant reductions in storage and improvement in performance. The framework and the objective functions are also highly flexible for customization and exploration of optimization strategies.

References

[1] S. Abu-Mahmeed, C. McCosh, Z. Budimli, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup. Scheduling tasks to maximize usage of aggregate variables in place. In *Intl. Conference on Compiler Construction (CC)*, pages 204–219, 2009.

[2] A. V. Aho, R. Sethi, J. D. Ullman, and M. S. Lam. *Compilers: Principles, Techniques, and Tools Second Edition*. Prentice Hall, 2006.

[3] C. Alias, F. Baray, and A. Darte. 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*, pages 73–82, 2007.

[4] S. G. Bhaskaracharya, U. Bondhugula, and A. Cohen. Automatic Storage Optimization for Arrays. *ACM Transactions on Programming Languages and Systems, accepted in 2015*.

[5] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

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

[7] P. Ghysels and W. Vanroose. Modeling the performance of geometric multigrid stencils on multicore computer architectures. *SIAM J. Scientific Computing*, 37(2):C194–C216, 2015.

[8] GNU. Gnu linear programming kit (glpk), 2010. https://www.gnu.org/software/glpk/.

[9] E. D. Greef, F. Catthoor, and H. D. Man. Array placement for storage size reduction in embedded multimedia systems. *Intl. Conf. on Application Specific Systems, Architectures, and Processors*, pages 66–75, 1997.

[10] E. D. Greef, F. Catthoor, and H. D. Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23(12):1811–1837, 1997.

[11] LabVIEW Compiler. NI LabVIEW Compiler: Under the Hood. http://www.ni.com/white-paper/11472/en.

[12] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[13] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, 2015.

[14] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 519–530, 2013.

[16] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[17] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Intl. conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 24–33, 1998.

[18] W. Thies, F. Vivien, J. Sheldon, and S. P. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 232–242, 2001.

[19] W. Thies, F. Vivien, and S. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007.

[20] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327, pages 299–302. Springer, 2010.

[21] D. Wilde and S. V. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Intl. Euro-Par Conference on Parallel Processing*, pages 389–397, 1996.