# Automatic Storage Optimization of Arrays in Affine Loop Nests

A Thesis

Submitted For the Degree of

**Doctor of Philosophy**

in Computer Science and Engineering

by

**Somashekaracharya G. Bhaskaracharya**


Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

July 2016

TO

*My family, friends and colleagues*

# ACKNOWLEDGEMENTS

I am deeply indebted to my advisor, Dr. Uday Bondhugula, for guiding me at every stage of my research work. It has truly been an honour to work with him. I hope I have justified the trust he invested in me when he agreed to be my advisor, despite the commitment to academic work only being of a part-time nature from my side. His constant support and optimism were major factors in helping me juggle both, academic research and office work at National Instruments, through the various ups and downs in these last five years. He was ever ready to discuss new ideas, however wild (I still remember a discussion on the storage optimization problem we had in the Dubai airport terminal, very late into the night, while waiting for our return flight to Bangalore). And almost invariably, I returned from such discussions with more ideas to work on. This work would, quite simply, not have been possible without his tremendous guidance. I offer my sincerest thanks!

I would like to express my gratitude to Dr. Albert Cohen for all his insightful comments and suggestions. Interactions with him, over email as well as in person every now and then, helped me immensely in refining the ideas presented here. His belief in their merit kept me inspired and motivated. I would also like to thank my organizational supervisor, Dr. Dinesh Nair, who gave me complete freedom in my research work.

I have greatly benefited through help from various other quarters. Firstly, I would like to thank my labmates Chandan, Roshan, Irshad, Vinay, Raviteja, Aravind, Thejas, Vinayak

for their generous assistance and suggestions on so many occasions, despite already being burdened with their own work. I am also very grateful to my friends and colleagues at NI who went out of their way in helping me pursue my academic goals to the fullest extent possible – to Anand, Gowrishankar, Rajanikanth for creating this opportunity for me; to Praveen, Subbaiah and Prashanth for their managerial support; and last but not the least, to my team-mates Nikhil, Bharath, Rakesh, Chethan and Ashwin. I would like to acknowledge the monetary assistance provided by NI towards my studies.

As the proverbial dwarf standing on the shoulders of giants, I owe a great deal to the authors of various tools (such as Clan, GLPK, ISL, Pet, Pluto etc) that I have used to implement the ideas presented in this work. Valuable feedback from several anonymous reviewers was also very helpful – my thanks to all these reviewers.

Finally, I would like to dedicate this work to my parents, G. S. Bhaskaracharya and H. K. Pushpa Latha, who have always encouraged me in all my pursuits; my thanks also to my brother and sister-in-law. The unflinching moral support at home helped me stay focused on my work.

# PUBLICATIONS BASED ON THIS THESIS

1. Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, *PolyGLoT: A Polyhedral Loop Transformation Framework for a Graphical Dataflow Language*, International conference on Compiler Construction (CC 2013), Rome, Italy, pages 123 - 143, March 2013.

2. Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, Albert Cohen, *Automatic Storage Optimization for Arrays*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol 38, issue 3, pages 11:1–11:23, April 2016.

3. Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, Albert Cohen, *SMO: An Integrated Approach to Intra-Array and Inter-Array Storage Optimization*, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), St.Petersberg, USA, pages 526 - 538, Jan 2016.

# ABSTRACT

Efficient memory usage is crucial for data-intensive applications as a smaller memory footprint ensures better cache performance and allows one to run a larger problem size given a fixed amount of main memory. The solutions found by existing techniques for automatic storage optimization for arrays in affine loop-nests, which minimize the storage requirements for the arrays, are often far from good or optimal and could even miss nearly all storage optimization potential. In this work, we present a new automatic storage optimization framework and techniques that can be used to achieve intra-array as well as inter-array storage reuse within affine loop-nests with a pre-determined schedule.

Over the last two decades, several heuristics have been developed for achieving complex transformations of affine loop-nests using the polyhedral model. However, there are no comparably strong heuristics for tackling the problem of automatic memory footprint optimization. We tackle the problem of storage optimization for arrays by formulating it as one of finding the right storage partitioning hyperplanes: each storage partition corresponds to a single storage location. Statement-wise storage partitioning hyperplanes are determined that partition a unified global array space so that values with overlapping live ranges are not mapped to the same partition. Our integrated heuristic for exploiting intra-array as well as inter-array reuse opportunities is driven by a fourfold objective function that not only minimizes the dimensionality and storage requirements of arrays required

for each high-level statement, but also maximizes inter-statement storage reuse.

We built an automatic polyhedral storage optimizer called SMO using our storage partitioning approach. Storage reduction factors and other results that we obtained from SMO demonstrate the effectiveness of our approach on several benchmarks drawn from the domains of image processing, stencil computations, high-performance computing, and the class of tiled codes in general. The reductions in storage requirement over previous approaches range from a constant factor to asymptotic in the loop blocking factor or array extents – the latter being a dramatic improvement for practical purposes.

As an incidental and related topic, we also studied the problem of polyhedral compilation of graphical dataflow programs. While polyhedral techniques for program transformation are now used in several proprietary and open source compilers, most of the research on polyhedral compilation has focused on imperative languages such as C, where the computation is specified in terms of statements with zero or more nested loops and other control structures around them. Graphical dataflow languages, where there is no notion of statements or a schedule specifying their relative execution order, have so far not been studied using a powerful transformation or optimization approach. The execution semantics and referential transparency of dataflow languages impose a different set of challenges. In this work, we attempt to bridge this gap by presenting techniques that can be used to extract polyhedral representation from dataflow programs and to synthesize them from their equivalent polyhedral representation. We then describe PolyGLoT, a framework for automatic transformation of dataflow programs that we built using our techniques and other popular research tools such as Clan and Pluto. For the purpose of experimental evaluation, we used our tools to compile LabVIEW, one of the most widely used dataflow programming languages. Results show that dataflow programs transformed using our framework are able to outperform those compiled otherwise by up to a factor of seventeen, with a mean speed-up of 2.30× while running on an 8-core Intel system.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

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

## 1.1 Automatic Storage Optimization

Automatic techniques that reduce the storage requirements are quite crucial for data-intensive applications. In several cases, a programmer is particularly interested in running a dataset while utilizing the entire main memory capacity of a system. In such cases, performance (execution time) is secondary. Storage optimization allows a programmer to run a larger problem size for a given main memory capacity. When using multiple applications,

it also allows more applications to fit in memory. In addition, storage optimization can also potentially improve performance as a direct result of a smaller memory footprint. It has also proved to be a critical optimization for domain-specific compilers. Image processing pipelines [RKBA+13] and stencil computations are two example domains where code generators rely on analysis to reduce the peak memory usage of the generated code. Compilers for functional languages with arrays, or dataflow languages with single-assignment semantics also need copy-avoidance to maximize memory reuse [AMMB+09].

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 that 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 [ASUL06]. Significant advances have been made in memory optimization for affine loop nests or its restricted forms [WR96, LF98, SCFS98, TVSA01, DSV05, ABD07]. However, we first show that a good memory optimization technique is still missing. The solutions found by existing works for several commonly encountered cases are far from good or optimal and could even miss nearly all storage optimization potential. Our method builds on the advances of the lattice-based memory contraction model [DSV05, ABD07], but approaches the problem in a new and very different way.

The storage optimization problem for arrays can be viewed as contracting the array along one or more dimensions to fixed sizes, or contracting along *directions* different from those along which the array is originally indexed. Thus, an approach to contraction can be viewed as one that finds: (1) good directions along which to contract (and the order in which to contract) in case the original ones are not good, and (2) the minimal sizes to which each of the chosen dimensions can be contracted. While the latter part was first comprehensively studied by Lefebvre and Feautrier [LF98], there is no heuristic available to obtain good solutions to the former. Choosing the right directions and their ordering impacts both the dimensionality of the resulting storage and the storage size. For example,

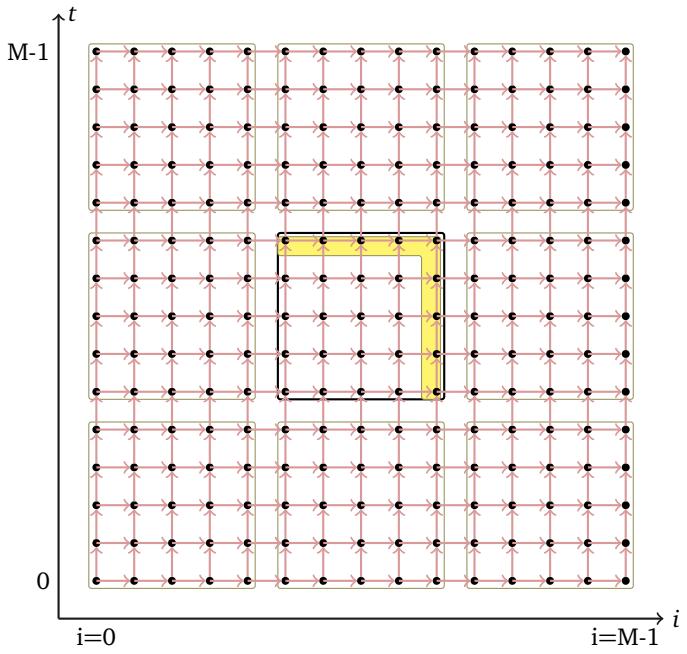



Figure 1.1: Storage requirement for the outlined tile (of size $T$) can be reduced to $2T-1$.

it can be the difference between say $N^2$, $2N$, and $N$ storage for what was originally an $N \times N$ array. Earlier approaches [LF98, DSV05] had either worked with the canonical (original) basis or assumed that the right directions would be provided by an oracle.

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

Consider the stencil computation with dependences $(1,0)$, $(0,1)$ in Figure 1.1. It corresponds to the tiled version of the code in Figure 3.1. For a given tile, only its top and right boundaries are live-out. As the primary objective behind tiling for locality is to exploit reuse in multiple directions while the data accessed fits in faster memory, live-out sets along two or more boundaries are common with tiling. In Figure 1.1, for a schedule that iterates row-wise within a tile, indexing the array along the canonical directions does not reduce storage, i.e., if $T$ is the tile size, $T^2$ storage per tile is needed. This solution corresponds to the canonical storage hyperplanes $(1,0)$ and $(0,1)$. The contraction factors obtained by Lefebvre and Feautrier [LF98] would just be $N$ along each of the two dimensions. None of the heuristics described in [DSV05, ABD07] find a different basis. If the array is partitioned along the hyperplane $(1,-1)$, i.e., if all points $(t,i)$ in the array such that $t-i = constant$ reuse the same memory location, the tile can be executed using a storage of just $2T-1$ cells. The storage buffer would finally hold the $2T-1$ live-out values. An access $A[t,i]$ will be transformed to an access $A[(t-i) \bmod (2T-1)]$, and this is also the optimal solution. The occupancy vector based approach of Strout et al. [SCFS98] does obtain this optimal storage, but it is designed for perfect loop nests with constant dependences, and its schedule-independent nature leads to sub-optimal solutions in general. The schedule-dependent approach we develop in this paper finds the optimal storage mapping in this case automatically, and works for general affine loop nests. Other dependence patterns or more complex tiling shapes can lead to non-trivial mappings that are very difficult to derive by hand.

In addition to reuse opportunities across elements of a single array, there may also exist some scope for storage reuse or across multiple arrays. Intra-array storage management deals with how an array written to by a particular statement is compacted and accessed. On the other hand, 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 [AMMB+09, Lab10, RKBA+13, MVB15].

```
1 for (t=1; t<=N; t++){
2   for (i=1; i<=N; i++)
3     P[i] = f(Q[i-1],Q[i],Q[i+1]);
4   for (i=1; i<=N; i++)
5     Q[i] = P[i];
6 }
7 for (i=1; i<=N; i++)
8   result += Q[i];
```

Figure 1.2: A stencil using a ping-pong buffer

Consider the code in Figure 1.2. 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 [LF98], Darte et al. [DSV05] 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 [LF98] or that of De Greef et al. [GCM97b] 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

```
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)];
```

(a) Figure 1.2 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)];
```

(b) After elimination of the dead code in Figure 1.3(a)

Figure 1.3: Storage optimization of ping-pong style 1-d stencil (from 2N to N+1)

Figure 1.2. 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 code shown in Figure 1.3(a) 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 (refer Figure 1.3(b)). This optimization opportunity is non-trivial to infer from the original code. Such a mapping not only leads to smaller storage, but also eliminates an unnecessary copy between the arrays while preserving semantics. The approach we present in this work is able to determine such mappings automatically. In the case of more realistic examples that employ 2-d or 3-d arrays, the

reduction is more prominent: from $2N^2$ to $N^2 + 2N$ for a code similar to Figure 1.2 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.

Our integrated approach for intra-array as well as inter-array storage optimization further builds on the notion of storage hyperplanes introduced earlier. Storage hyperplanes, in the context of inter-array storage optimization, 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. For example, the hyperplane that enables the storage optimization in Figure 1.3(a) is $(i-t) = (-1,1)\cdot(t,i)^T$. 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 essence, our contributions towards the problem of automatic storage optimization for arrays can be summarized as follows.

- We describe a new technique for storage optimization while casting the latter as an array space partitioning problem, where each partition uses a single memory location. We then formulate an ILP problem solvable using a greedy heuristic whose objective takes into account the dimensionality of the mapping as well as the extent along each dimension.

- We further develop an integrated approach to intra-statement as well as inter-statement storage optimization by generalizing the principle of storage partitioning for a unified global array space. We then present a greedy solution for finding statement-wise storage hyperplanes. The greedy objective is not only based on the dimensionality of the mapping and the extents along each dimension but also factors in the inter-statement storage reuse considerations.

- We implement and evaluate our technique on various domain-specific benchmarks and demonstrate reductions in storage requirement ranging from a constant factor to asymptotic in the extents of the original array dimensions or loop blocking factors.

Finally, the techniques that we propose for storage optimization in affine loop-nests, by exploiting intra-array and inter-array reuse opportunities, have been built into SMO [SMO16], an open source tool which was developed as part of this work.

## 1.2 Polyhedral Compilation of Dataflow Programs

As a somewhat related topic, we also study the problem of polyhedral compilation of dataflow programs. Many computationally intensive scientific and engineering applications that employ stencil computations, linear algebra operations, image processing kernels, etc. lend themselves to polyhedral compilation techniques [ASUL06, Bas]. Such computations exhibit certain properties that can be exploited at compile time to perform parallelization and data locality optimization.

Typically, the first stage of a polyhedral optimization framework consists of polyhedral extraction. Specific regions of the program that can be represented using the polyhedral model, typically affine loop nests, are analyzed. Such regions have been termed Static Control Parts (SCoPs) in the literature. Results of the analysis include an abstract mathematical representation of each statement in the SCoP, in terms of its iteration domain, schedule, and array accesses. Once dependences are analyzed, an automatic parallelization and locality optimization tool such as Pluto [Plu] is used to perform high-level optimizations. Finally, the transformed loop nests are synthesized using a loop generation tool such as CLooG [Bas04].

Regardless of whether an input program is written in an imperative language, a dataflow language, or using another paradigm, if a programmer does care about performance, it is important for the compiler not to ignore transformations that yield significant performance gains on modern architectures. These transformations include, for example, ones that enhance locality by optimizing for cache hierarchies and exploiting register reuse or those

that lead to effective coarse-grained parallelization on multiple cores. It is thus highly desirable to have techniques and abstractions that could bring the benefit of such transformations to all programming paradigms.

There are many compilers, both proprietary and open-source that now use the polyhedral compiler framework [GZA+11, MVW+11, BGDR10, Plu]. Research in this area, however, has predominantly focused on imperative languages such as C, C++, and Fortran. These tools rely on the fact that the code can be viewed as a sequence of statements executed one after the other. In contrast, a graphical dataflow program consists of an interconnected set of nodes that represent specific computations with data flowing along edges that connect the nodes, from one to another. There is no notion of a statement or a mutable storage allocation in such programs. Conceptually, the computation nodes can be viewed as consuming data flowing in to produce output data. Nodes become ready to be 'fired' as soon as data is available at all their inputs. The programs are thus inherently parallel. Furthermore, the transparency with respect to memory referencing allows such a program to write every output data value produced to a new memory location. Typically, however, copy avoidance strategies are employed to ensure that the output data is *inplace* to input data wherever possible. Such *inplaceness* decisions can in turn affect the execution schedule of the nodes.

The polyhedral extraction and code synthesis for dataflow programs, therefore, involves a different set of challenges to those for programs in an imperative language such as C. In this work, we propose techniques that address these issues. Furthermore, to demonstrate their practical relevance, we describe an automatic loop transformation framework that we built for the LabVIEW graphical dataflow programming language, which uses all of these techniques. Our contributions regarding the polyhedral compilation of graphical dataflow programs are as follows.

- We provide a specification of parts of a dataflow program that lends itself to the abstract mathematical representation of the polyhedral model.
- We describe a general approach for extracting the polyhedral representation for such a dataflow program part and also for the inverse process of code synthesis.

- We present an experimental evaluation of our techniques for LabVIEW and comparison with the LabVIEW production compiler.

Chapter 2 provides the necessary background on the polyhedral model, the successive modulo technique of Lefebvre and Feautrier [LF98], a brief introduction to dataflow languages (LabVIEW, in particular) and introduces the notation used later. Chapters 3 and 4 present the details of our storage optimization scheme for exploiting intra-array and inter-array reuse respectively, along with related work and various examples drawn from real-world applications. Chapter 5 reports results obtained using an implementation of our automatic array optimization techniques. Most of the content in these three chapters has been published in [BBC16a] and [BBC16b]. Chapter 6 is based on our work published in [BB13]. It describes PolyGLoT, an auto-transformation framework for the LabVIEW graphical dataflow programming languages, and also provides a detailed discussion of the techniques that we developed to build this framework. An experimental evaluation of the same is reported in Section 6.4. Finally, our conclusions are presented in chapter 7.

# CHAPTER 2

# BACKGROUND

This chapter provides the background and notation required for the techniques that we describe in the later chapters for storage optimization and polyhedral compilation of dataflow programs.

## 2.1 Affine Hyperplane

**Definition 1.** *The set of all vectors $\vec{v} \in \mathbb{Z}^n$ such that $\vec{h}.\vec{v} = k$ constitute an affine hyperplane.*

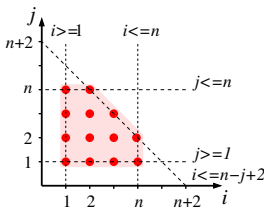
Different constant values for $k$ generate different parallel instances of the hyperplane which is usually characterized by the vector, $\vec{h}$, normal to it.

## 2.2 Polyhedral Model

### 2.2.1 Overview of the Polyhedral Model

The polyhedral model provides an abstract mathematical model to reason about program transformations. Consider a program part that is a sequence of statements with zero or more loops surrounding each statement. The loops may be imperfectly nested. The

```
1 for (i=1; i<=n; i++)
2   for (j=1; j<=n; j++)
3     if (i <= n-j+2)
4       S1;
```




Iteration domain of S1

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ -n \\ -n-2 \end{pmatrix}$$

Iteration domain of S1

Figure 2.1: Polyhedral representation of a loop nest in geometric and linear algebraic form

```
1 for (i=1; i<=n-1; i++)
2   for (j=i+1; j<=n; j++)
3 /*S1*/ c[i][j] = a[j][i] / a[i][i];
4   for (j=i+1; j<=n; j++)
5     for (k=i+1; k<=n; k++)
6 /*S2*/ a[j][k] -= c[i][j] * a[i][k];
```

(a) Original code

$$\theta_{S1}(i,j) = (i,0,j,0)$$
$$\theta_{S2}(i,j,k) = (i,1,j,k)$$

(b) Initial schedule

$$\theta_{S_1}(i,j) = (i+j,j,0)$$
$$\theta_{S_2}(i,j,k) = (i+j,j,1,k)$$

(c) New schedule

```
1 for (t1=3; t1<=2*n-1; t1++)
2   for (t2=ceild(t1+1,2); t2<=min(n,t1-1); t2++)
3     c[t1-t2][t2] = a[t2][t1-t2] / a[t1-t2][t1-t2];
4     for (t3=t1-t2+1; t3<=n; t3++)
5       a[t2][t3] -= c[t1-t2][t2] * a[t1-t2][t3];
```

(d) Transformed code

Figure 2.2: Example of input code, the corresponding original schedule, a new schedule and transformed code.

dynamic instances of a statement $S$, are represented by the integer points of a polyhedron whose dimensions correspond to the enclosing loops. The set of dynamic instances of a statement is called its *iteration domain*, $D$. It is represented by the polyhedron, defined by a conjunction of affine inequalities that involve the enclosing loop iterators and global parameters. Each dynamic instance is uniquely identified by its *iteration vector*, i.e., the vector $\vec{i}_S$ of enclosing loop iterator values. Figure 2.1 shows the polyhedral representation of a loop nest in its geometric and linear algebraic form.

**Schedules.** Each statement, or more precisely its domain, has an associated schedule, which is a multi-dimensional affine function mapping each integer point in the statement's domain to a unique time point that determines when it is to be executed. Code generated from the polyhedral representation scans integer points corresponding to all statements

globally in the lexicographic order of the time points they are mapped to. For example, $\theta_S(i,j,k) = (i+j, j, k)$ is a schedule for a 3-d loop nest with original loop indices $i$, $j$, $k$. Changing the schedule to $(i+j, k, j)$ would interchange the two inner loops. Figure 2.2 shows a code, its schedule as extracted from the input program, a new schedule and code generated with the new schedule. The new schedule fuses the $j$ loops of both the statements while skewing the outermost loop with respect to the second outermost one. The reader is referred to [Bas] for more details on the polyhedral representation.

The initial schedule that is extracted, corresponding to the original execution order, is referred to as an *identity schedule*, i.e., if it is not modified, code generation will lead to the same code as the one from which the representation was extracted. A dimension of the multi-dimensional affine scheduling function is called a *scalar dimension* if it is a constant. In Figure 2.2(b), the second dimension of both statements' schedules are scalar dimensions. In the schedules listed in Figure 2.2(c), the third dimension is a scalar one. Polyhedral optimizers have models to pick a suitable schedule among valid ones. A commonly used model that minimizes dependence distances in the transformed space [BBK+08], thereby optimizing locality and parallelism simultaneously, is implemented in Pluto [Plu].

## 2.3 Farkas' Lemma

Several polyhedral techniques rely on the application of the affine form of the Farkas' lemma [Sch86, Fea92a].

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

$$\psi(\vec{x}) = \lambda_0 + \Sigma_{(k=1)}^{(k=s)} \lambda_k \left(\vec{a_k}.\vec{x} + b_k\right), \lambda_k \geq 0. \tag{2.1}$$

The $\lambda_k$s are known as Farkas multipliers.

## 2.4 Successive Modulo Technique

Lefebvre and Feautrier [LF98] proposed a storage optimization technique which they referred to as partial data expansion. A given static control program is subjected to array dataflow analysis and then converted into functionally equivalent single-assignment code so that all the artificial dependences (output and anti) are eliminated. The translation to single-assignment code involves rewriting the program so that each statement $S$ writes to its own distinct array space $A_S$, which has the same size and shape as that of the iteration domain of $S$. Without any loss of generality, if we assume that the loop indices are non-negative, then $\vec{i_S}$ writes to $A_S[\vec{i}]$. This process of expanding the array space is known as *total data expansion*. A schedule $\theta$ 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 live, 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 that 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 $\theta(\vec{i_S}) \preceq \theta(L(\vec{j_S}))$ and $\theta(\vec{j_S}) \preceq \theta(L(\vec{i_S}))$ i.e., iff the corresponding array elements are simultaneously live under the given schedule $\theta$.*

The conflict set $CS$ is the set of all pairs of conflicting indices. It can be specified as $CS = \{(\vec{i}, \vec{j}) \mid \vec{i} \bowtie \vec{j}\}$. In accordance with the above definition, the conflict relation $\bowtie$ is symmetric and non-reflexive. Partial expansion is performed iteratively with each statement being considered once at every depth of the surrounding loop nest. The *contraction modulo $e_p$* (or expansion degree as Lefebvre and Feautrier [LF98] refer to it), along the axis of the array space which corresponds to the loop at depth $p$, is computed as follows.

Suppose $DS$ is the set of differences of indices which conflict, i.e., $DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$. Similarly, let $DS_p = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j} \wedge \vec{i} \succ \vec{j} \wedge (i_x = j_x \forall x < p)\}$. If $\vec{b}$ is the lexicographic maximum of $DS_p$, the contraction modulo is given by $e_p = b_p + 1$, where $b_p \hat{u}_p$ is the component of $\vec{b}$ along the axis $i_p$, with $\hat{u}_p$ representing the unit vector along the same axis. In essence, the contraction modulo $e_p$ represents the degree of contraction along that axis. The final storage mapping is obtained by converting it into a modulo mapping so that the statement instance $\vec{i}_S$ writes to $A_S[\vec{i} \bmod \vec{e}]$, where $\vec{e} = (e_0, e_1, \ldots, e_{n-1})$. This method to determine the contraction moduli will hereafter be referred to as the *successive modulo technique*.

## 2.5 Rectangular Hull for Inter-Array Reuse

Lefebvre and Feautrier [LF98] 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 corresponding to the statements $S_i$ and $S_j$ implies that the two statements cannot write to the same data structure — the prescribed shared data structure is nothing but the 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 that can write to such a shared data structure. Hereafter, we refer to this technique as the *rectangular hull method*.

## 2.6 LabVIEW – Language and Compiler

LabVIEW is a graphical, dataflow programming language from National Instruments Corporation (NI) that is used by scientists and engineers around the world. Typically, it is used for implementing control and measurement systems, and embedded applications.

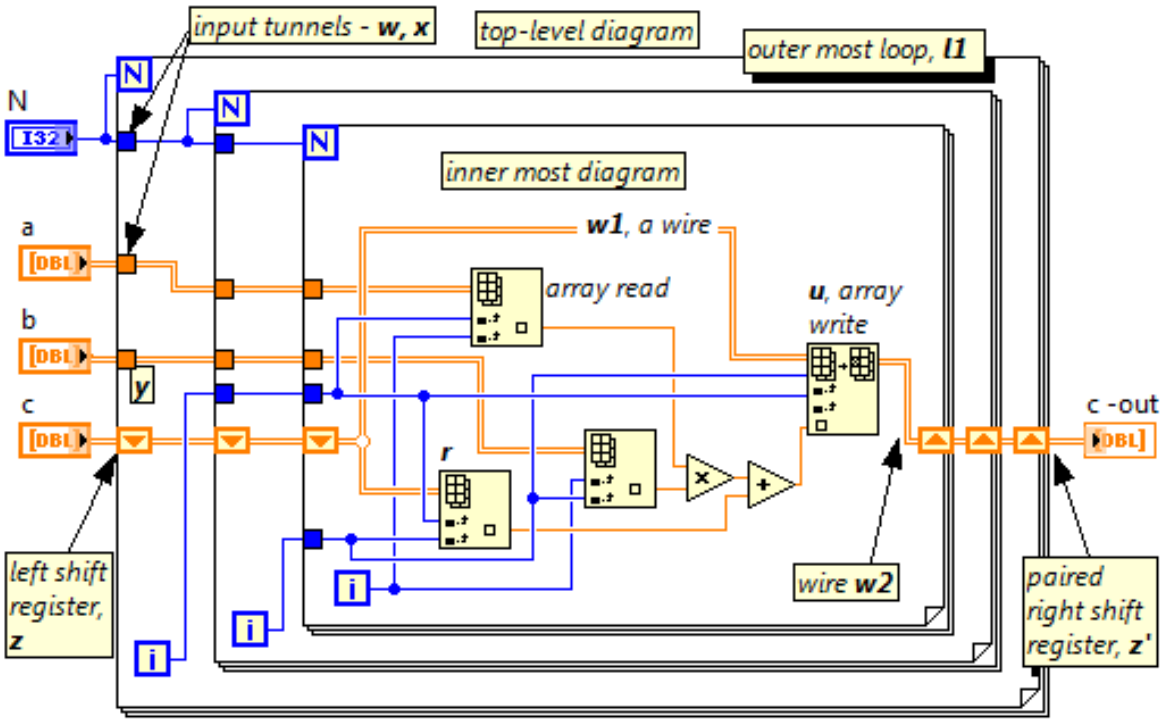



Figure 2.3: *matmul* in LabVIEW

The language itself, due to its graphical nature, is referred to as the G language. A LabVIEW program called a Virtual Instrument (VI) consists of a front panel (the graphical user interface) and a block diagram, which is the graphical dataflow diagram. Instead of textual statements, the program consists of specific computation nodes. The flow of data is represented by a *wire* that links the specific output on a source node to the specific input on a sink node.

The block diagram of a LabVIEW VI for matrix multiplication is shown in Figure 2.3. LabVIEW for-loops are unit-stride for-loops with zero-based indexing. A loop iterator node in the loop body (the [i] node) produces the index value in any iteration. A special node on the loop boundary (the N node) receives the upper loop bound value. The input arrays are provided by the nodes $a, b$ and $c$. The output array is obtained at the node named $c - out$. The color of the wire indicates the type of data flowing along it e.g. blue for integers, orange for floats. Double lines are indicative of arrays.

Loop nodes act as special nodes that enclose the dataflow computation that is to be

executed in a loop. Data that is only read inside the loop flows through a special node on the boundary of the loop structure called the *input tunnel*. A pair of boundary nodes called the *left* and *right shift registers* are used to represent loop-carried dependence. Data flowing into the right shift register in one iteration flows out of the left shift register in the subsequent iteration. The data produced as a result of the entire loop computation flows out of the right shift register. Additionally, some boundary nodes are also used for the loop control. In addition to being inherently parallel because of the dataflow programming paradigm, LabVIEW also has a parallel for loop construct that can be used to parallelize the iterative computation [BDYF10].

The LabVIEW compiler first translates the G program into a Data Flow Intermediate Representation (DFIR) [Lab10]. It is a high-level, hierarchical and graph-based representation that closely corresponds to the G code. Likewise, we model the dataflow program as being conceptually organized in a hierarchy of diagrams. It is assumed that the diagrams are free of dead-code.

## 2.7 An Abstract Model of Dataflow Programs

Suppose N is the set of computation nodes and W is the set of wires in a particular diagram. Each diagram is associated with a directed acyclic graph (DAG), $G = (V, E)$, where $V = N \cup W$ and $E = E_N \cup E_W$. $E_N \subseteq N \times W$ and $E_W \subseteq W \times N$. Essentially, $E_N$ is the set of edges that connect the output of the computation nodes to the wires that will carry the output data. Likewise, $E_W$ is the set of edges that connect the input of computation nodes to the wires that propagate the input data. We follow the convention of using small letters $v$ and $w$ to denote computation nodes and wires respectively. Any edge $(v, w)$ represents a particular output of node $v$ and any edge $(w, v)$ represents a particular input of node $v$. So, the edges correspond to memory locations. The wires serve as copy nodes, if necessary. Figure 2.4 shows the DAG in the block diagram of the LabVIEW matrix multiplication program. In this abstract model, the gray nodes are wires. The 4 source nodes (N, a, b, c), the sink node (c-out) and the outermost loop are represented as the 6 blue nodes. Directed

edges represent the connections from inputs/outputs of computation nodes to the wires, e.g. data from source node N flows over a wire into two inputs of the loop node. Hence the two directed edges from the corresponding wire node.

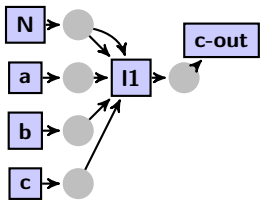



Figure 2.4: DAG of the top-level diagram of *matmul*.

For every $n \in N$ that is a loop node, it is associated with a DAG, $G_n = (V_n, E_n)$ which corresponds to the dataflow graph describing the loop body. The loop inputs and outputs are represented as source and sink vertices. The former have no incoming edges, whereas the latter have no outgoing edges. Let $I$ and $O$ be the set of inputs and outputs. Furthermore, a loop output vertex may be paired with a loop input vertex to signify a loop-carried data dependence i.e., data produced at the loop output in one iteration flows out of the input for the next iteration (Fig 2.3).

**Loop Inputs and Outputs.** Data flowing into and out of a loop is classified as either *loop-invariant input data* or *loop-carried data*. Loop-invariant input data is that which is only read in every iteration of the loop. Let $Inv$ be the set of loop-invariant data inputs to the loop. The LabVIEW equivalent for such an input is an input tunnel. In Figure 2.3, for the outermost loop $l_1$, $Inv = \{w, x, y\}$. Loop-carried data is that which is part of a loop-carried dependence inducing dataflow. The paired loop inputs and outputs represent such a dependence. Let $ICar, OCar$ be sets of these loop inputs and outputs. The loop-carried dependence is represented by the one-to-one mapping $lcd : OCar \rightarrow ICar$. The LabVIEW equivalent for such a pair are the left and right shift registers. In Figure 2.3, for loop $l_1, ICar = \{z\}, OCar = \{z'\}, (z, z') \in lcd$.

### 2.7.1 Inplaceness

In accordance with the referential transparency of a dataflow program, each edge could correspond to a new memory location. Typically, however, a copy-avoidance strategy may be used to reuse memory locations. For example, consider the array element write node $u$ in Figure 2.3, and its input and output wires, $w_1$ and $w_2$. The output array data flowing along $w_2$ could be stored in the same memory location as the input array data flowing along $w_1$. The output data can be *inplace* to the input data. The *can-inplace* relation $(w_1, u) \rightsquigarrow (u, w_2)$ is said to hold.

In general, for any two edges $(x, y)$ and $(y, z)$, $(x, y) \rightsquigarrow (y, z)$ holds iff the data inputs or outputs that these edges correspond to *can* share the same memory location (regardless of whether a specific copy-avoidance strategy chooses to reuse the memory location or not). The can-inplace relation is an equivalence relation. A path $\{x_1, x_2, \ldots, x_n\}$ in a graph $G = (V, E)$, such that $(x_{i-1}, x_i) \rightsquigarrow (x_i, x_{i+1})$ for all $2 \leq i \leq n-1$, is said to be a *can-inplace* path. Note that by definition, the can-inplace relation $(w_1, v) \rightsquigarrow (v, w_2)$ implies that the node $v$ can overwrite the data flowing over $w_1$. In such a case, we say that the relation $v \times w_1$ holds. However, the can-inplace relation $(v_1, w) \rightsquigarrow (w, v_2)$ does not necessarily imply such a destructive operation as the purpose of a wire is to propagate data, not to modify it.

Suppose $<_s$ is a binary relation on $V$ which specifies a total ordering of the computation nodes. The relation $<_s$ specifies a *valid* execution schedule iff $(v_1 <_s v_2)$ implies that there does not exist a directed path in graph $G$, from $v_2$ to $v_1$ for any $v_1, v_2 \in V$ i.e., the schedule respects all dataflow dependences. As we shall see later, the problem of scheduling the computation nodes is closely related to inplaceness. Memory reuse due to copy-avoidance can create additional dependences. A conjunction of scheduling relations $\bigwedge(v_1 <_s v_2)$ is said to be *consistent* with a conjunction of can-inplace relations $\bigwedge((x, y) \rightsquigarrow (y, z))$, for $x, y, z \in N \cup V$, iff such a schedule does not violate the dependences imposed by such an inplaceness choice.

**Array Accesses.** In Figure 2.3, the array read access is a node that takes in an array and the access index values to produce the indexed array element value. The array write access, takes the same set of inputs and the value to be written to produce an array with the indexed element overwritten. We model the array read and write accesses similarly. Notice that the output array of an array write, $v$, need not be inplace to the input array flowing through a wire $w_1$. If it is, then the array write performs a destructive update and we represent that using the relation $v \times w_1$.

# CHAPTER 3

# INTRA-ARRAY STORAGE OPTIMIZATION

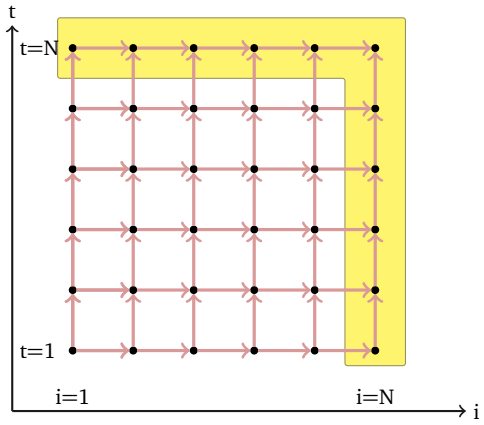
In this chapter, we present all the details of our storage partitioning approach for achieving intra-array storage optimization.

## 3.1 A Simple Example

The successive modulo technique, described in Section 2.4, is quite versatile, scalable and also parametric. However, the eventual modulo storage mapping obtained does not always lead to minimal storage requirements. Consider the static control loop nests in Figure 3.1(a). The producer loop nest is already in single-assignment form where each statement instance $S(t,i)$ writes to its own distinct memory cell $A[t,i]$ so that the array space $A$ has the same size and shape as the iteration domain of statement $S$. Suppose the schedule determined is $\theta(t,i) = (t,i)$. There are some values computed by statement $S$ which are live even after all its instances have been executed. These live-out values reside in the set of memory cells, $\{(t,i) \mid (t,i) \in A \wedge (i = N) \vee (t = N)\}$. As a result, the conflict set $CS$ is made up of conflicts not only due to the uniform lifetimes of the non-live-out values but also due to the non-uniform lifetimes of the live-out values. Specifically, the

```
1 // the producer loop
2 for (t=1; t<=N; i++)
3    for (i=1; i<=N; i++)
4 /*S*/ A[t,i] = A[t,i-1] + A[t-1,i];
5
6 // the consumer loop
7 for (i=1; i<=N; i++)
8    result = result + A[i,N] + A[N,i];
```

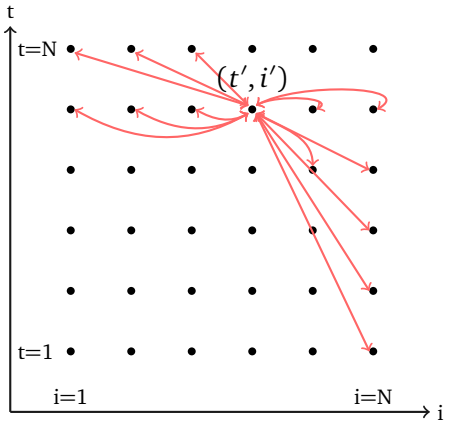
(a) Iteration domain of S is shown in Figure 3.1(b).




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

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

(c) Conflict set specification




(d) A geometrical representation of conflicts

Figure 3.1: The geometrical representation in Figure 3.1(d) shows the array space $A$ written to by statement $S$ in the code snippet shown in Figure 3.1(a). The red double-headed arrows in Figure 3.1(d) denote the various conflicts associated with the array index $(t', i')$.

array index associated with a live-out value conflicts with the array index associated with any value computed later in the schedule. So, the resulting conflict set $CS$, of pairs of conflicting indices $(t,i) \bowtie (t',i')$, is a union of convex polyhedra characterized by the constraints shown in Figure 3.1(c).

The conflict relation is, strictly speaking, symmetric. For brevity, the constraints in

Figure 3.1(c) represent a conflict between a pair of conflicting indices only once, effectively treating it as an unordered pair. The first two disjuncts in Figure 3.1(c) together represent conflicts due to the flow dependence $(1,0)$, which is also the maximum utility span of any non-live-out value (refer Figure 3.1(b)). The last disjunct expresses the conflicts due to the live-out values. The geometrical representation in Figure 3.1(d) shows the array space $A$ written to by statement $S$ in the code snippet shown in Figure 3.1(a). The red double-headed arrows in Figure 3.1(d) denote the various conflicts associated with the array index $(t', i')$. Please note that often, the last read of the value computed by one statement instance and the write by another instance of the same statement occur at the same logical time. Hereafter, in such scenarios, we do not treat the associated indices as conflicting since they can be mapped to the same memory cell e.g. in Figure 3.1(c), $(t', i')$ and $(t'+1, i')$ do not conflict.

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

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

## 3.2 Storage Hyperplanes and Conflict Satisfaction

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

**Definition 3.** *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 satisfied only if the array indices involved are mapped to different partitions.

The successive modulo technique can also be understood through this notion of conflict satisfaction. Consider again loop nest in Figure 3.1. As explained earlier, the contraction modulo $e_0 = N$ is due to the conflict $(1,N) \bowtie (N,N)$. This is equivalent to the hyperplane $(1,0)$ partitioning the array space into $N$ partitions. Clearly, the conflicting indices $(1,N)$ and $(N,N)$ are mapped to different partitions, thus satisfying the conflict. The hyperplane $(1,0)$ satisfies all the conflicts represented by the second and third disjuncts in Figure 3.1(c). The conflicts specified by the first disjunct are not satisfied as the conflicting indices get mapped to the same partition. However, these conflicts are satisfied at loop-depth $p = 1$. This can be seen as the hyperplane $(0,1)$ further partitioning each of the $N$ partitions obtained earlier into $N$ distinct sub-partitions. As a result, the conflicting indices in the conflicts that were not satisfied at the previous level get mapped to different partitions. In essence, the successive modulo approach can also be understood as conflict satisfaction being performed by successively partitioning the array space using a series of storage hyperplanes.

The dimensionality of the array space is a loose upper bound on the number of storage hyperplanes required to satisfy all the conflicts. This is because, in the trivial case, the hyperplanes could simply correspond to those which have the canonical axes as their normals. In fact, this is exactly how the modulo storage mapping is determined in the successive modulo technique. In Figure 3.1, all the conflicts were satisfied using the two canonical hyperplanes, $(1,0)$, $(0,1)$. However, the mapping $A[t,i] \rightarrow A[(i-t) \bmod (2N-1)]$, which is better than the resulting solution not only in terms of the storage size required but

also in terms of its dimensionality, shows that it is possible to satisfy all the conflicts in the conflict set (Figure 3.1(c)) using just one storage hyperplane. Generally, the choice of partitioning hyperplanes affects both the dimensionality as well as the storage requirements of the resulting modulo storage mapping.

## 3.3 A Partitioning Approach

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

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

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

### 3.3.1 Conflict Set Specification

The conflict set can be specified as a union of convex polyhedra, also called *conflict polyhedra*, e.g., the disjunction in Figure 3.1(c). Each integer point in a conflict polyhedron represents a particular conflict. The symmetric property of the conflict relation can be

used to simplify the conflict set significantly. Consider a 1-d array space $A$ where all array indices conflict with all other indices. The conflict set $CS$ is then the set of ordered pairs $(i, i')$ such that $i \bowtie i'$ holds. A conflict relation $1 \bowtie 2$ can be encoded as the integer point $(1,2)$ in the conflict polyhedron $\{i < i' \mid i, i' \in A\}$. Strictly speaking though, if all conflict relations are to be represented, due to the symmetry, another conflict polyhedron $\{i > i' \mid i, i' \in A\}$ would be required to accommodate the conflict relation $2 \bowtie 1$. However, satisfying the conflict $1 \bowtie 2$ implies that $2 \bowtie 1$ is also satisfied as both of them represent the same pair of indices. The second conflict polyhedron is, in effect, redundant in the conflict set. Hereafter, we 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. There may be multiple ways to specify a conflict set as a union of conflict polyhedra. Therefore, we adhere to 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 than that for the index $\vec{i}$ according to the given schedule. Furthermore, the conflict set specification is minimal in the sense that no two conflict polyhedra exist in the union such that their union is itself convex.

## 3.4 Finding a Storage Hyperplane

In the scenario when the conflict set is empty to begin with, the optimal allocation is to contract the array down to a single scalar variable. Storage hyperplanes only need to be found when the conflict set is non-empty. Suppose there are $l$ conflict polyhedra $K_1, K_2, \ldots, K_l$ so that the conflict set $CS = \cup_{i=1}^{i=l} K_i$. Consider a pair of conflicting indices $\vec{s}$ and $\vec{t}$. By Definition 3, a hyperplane $\vec{\Gamma}$ satisfies this conflict if $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \neq 0$. This can be expressed by the disjunction:

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

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

vector of program parameters. Such a bound has been used in [Fea92a] and in [BBK+08], although in different contexts. Additionally, as the conflict relation is symmetric, the upper bound is applicable to the absolute value of the conflict difference $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t})$. So, the bounding constraints can be expressed as follows:

$$-\left(\vec{u}.\vec{P} + w\right) \;\leq\; \left(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\right) \;\leq\; \left(\vec{u}.\vec{P} + w\right). \tag{3.1}$$

### 3.4.1 Encoding Satisfaction with Decision Variables

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

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

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

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

$$\begin{aligned} & \left(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\right) \geq 1 - \left(1 - x_{1i}\right)\left(\vec{u}.\vec{P} + w + 1\right) \\ \wedge\quad & \left(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\right) \leq -1 + \left(1 - x_{2i}\right)\left(\vec{u}.\vec{P} + w + 1\right). \end{aligned} \tag{3.2}$$

By definition, $x_{1i}$ and $x_{2i}$ cannot be simultaneously equal to 1. Such a scenario would mean that the constraints in the above conjunction would contradict each other. However, if $x_{1i} = 1$ and $x_{2i} = 0$, then the first conjunct degenerates into the conflict satisfaction constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq 1$ whereas the second conjunct is reduced to the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \leq (\vec{u}.\vec{P} + w)$, which is implied by the bounding constraints (3.1). Similarly, if $x_{2i} = 1$ and $x_{1i} = 0$, the first conjunct becomes $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}) \geq -(\vec{u}.\vec{P} + w)$ which is again implied by the bounding constraints (3.1). 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 entire conjunction boils down to the bounding constraints (3.1), which must always hold, regardless of whether all or a few of the conflicts in the conflict polyhedron are satisfied.

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

$$\eta = \Sigma_{i=1}^{i=l} \left( x_{1i} + x_{2i} \right). \tag{3.3}$$

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

### 3.4.2 Linearizing the Constraints

The storage hyperplane $\vec{\Gamma}$ should be such that the bounding constraints (3.1) hold at every integer point $\vec{v}$ in a conflict polyhedron. Each conjunct in the bounding constraints can be rewritten to be in the form $\psi(\vec{v}) \geq 0$ where $\psi(\vec{v})$ is affine. By the Farkas' lemma (2.1), the affine form can be equated to a non-negative linear combination of the faces of the conflict polyhedron. The loop variables can then be eliminated by equating their respective coefficients to obtain an equivalent set of linear inequalities involving only the coefficients, some of which are the Farkas' multipliers. However, the same procedure cannot be repeated for the decision constraints in (3.2) as neither of the two conjuncts can be rewritten in the form $\psi(\vec{v}) \geq 0$ (refer (2.1)). The coefficients of $\vec{P}$ in both conjuncts are products of a decision variable and $\vec{u}$'s coefficients, and similarly $x_{1i}w$ and $x_{2i}w$ are non-linear. Therefore, the decision constraints in (3.2) cannot be linearized using the Farkas' lemma.

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

$$\begin{aligned} & \left(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\right) \leq \left(\vec{u}.\vec{P} + w\right) \leq \left(c\vec{P} + c\right) \\ \wedge \quad & -\left(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\right) \leq \left(\vec{u}.\vec{P} + w\right) \leq \left(c\vec{P} + c\right). \end{aligned} \tag{3.4}$$

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. Each individual constraint in (3.4) can be treated using Farkas' lemma to obtain a set of equivalent linear inequalities after eliminating the loop variables. Due to transitivity, $(c\vec{P} + c)$ is also an upper bound on $|\ \vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\ |$.

The decision constraints in (3.2) were formulated such that if either $x_{1i}$ or $x_{2i}$ is equal to 1, then one of the conjuncts degenerates into a conflict satisfaction constraint while the other into one of the bounding constraints in (3.1), which specify $(\vec{u}.\vec{P} + w)$ as an upper bound on $|\ \vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}\ |$. Now, along similar lines, an alternative set of decision constraints

can be formulated as follows:

$$\left(\vec{\Gamma}.\vec{s}-\vec{\Gamma}.\vec{t}\right) \geq 1-\left(1-x_{1i}\right)\left(c\vec{P}+c+1\right)$$
$$\wedge \quad \left(\vec{\Gamma}.\vec{s}-\vec{\Gamma}.\vec{t}\right) \leq -1+\left(1-x_{2i}\right)\left(c\vec{P}+c+1\right). \tag{3.5}$$

The key is that $c$ is a constant, and it now makes the conflict satisfaction constraints amenable to linearization through application of Farkas' lemma. Although the variables $\vec{u}$ and $w$ still feature in the expanded set of bounding constraints in (3.4), the decision constraints in (3.5) are now devoid of them. Note that $c\vec{P}+c$ has been substituted for $\vec{u}\vec{P}+w$ in (3.2) alone to obtain (3.5). The difference between (3.2) and (3.5) is only with respect to the upper and lower bounds that are imposed on $(\vec{\Gamma}.\vec{s}-\vec{\Gamma}.\vec{t})$ when one of the binary decision variables $x_{1i}$, $x_{2i}$ is equal to 1 or when both are equal to 0. It still holds that $x_{1i}$ and $x_{2i}$ cannot simultaneously be equal to 1. The bound $c\vec{P}+c$ only has to be sufficiently large to bound the conflict difference — it need not be tight and it does not influence the objectives we will propose and the solutions obtained in any way.

### 3.4.3 A Greedy Double-Objective

The resulting ILP system consists of the constraints obtained due to the expanded set of bounding constraints in (3.4), the revised decision constraints in (3.5) and also the constraint on $\eta$ given by (3.3). Such constraints are derived for each of the $l$ conflict polyhedra. The greedy approach is to determine a storage hyperplane $\vec{\Gamma}$ such that the estimated number of conflict polyhedra $\eta$, all of whose conflicts are satisfied, is maximized. This affects the dimensionality of the storage mapping which is eventually obtained.

Another factor that needs to be considered while determining the storage hyperplanes is the storage size of the resulting modulo storage mapping. The storage size of a modulo storage mapping determined using the successive modulo technique is computed as the product of the contraction moduli. In the successive modulo technique, the contraction moduli are computed along the canonical bases. Essentially, the canonical bases also serve as the storage hyperplane normals. In general though, the storage hyperplane normals

may not necessarily correspond to the canonical bases. However, the modulo can still be computed based on the maximum conflict difference ($\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}$), which is essentially the maximum component of any conflict difference ($\vec{s} - \vec{t}$) along the normal of the hyperplane $\vec{\Gamma}$. Since the contraction modulo is 1 plus the maximum conflict difference, the greater the maximum conflict difference, the more storage size required for the resulting storage mapping. Therefore, in order to minimize the storage size, another objective in solving the ILP system is to minimize the maximum conflict difference ($\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}$). As ($\vec{u}.\vec{P} + w$) is an upper bound on ($\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t}$), this can be done by minimizing ($\vec{u}.\vec{P} + w$).

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

$$minimize_{\prec} \left\{ \eta', u_1, u_2, \ldots, u_p, w \right\}. \tag{3.6}$$

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

Conflict satisfaction is the primary issue involved in partitioning. So, the objective gives minimization of $\eta'$ precedence over that of ($\vec{u}.\vec{P} + w$). As we shall see later, for scenarios such as the one in Figure 3.1(a), this ensures that a hyperplane which satisfies all conflicts at once is given precedence over a hyperplane which leaves some conflicts unsatisfied even if the contraction modulo for the latter is smaller than that for the former.

## 3.5 Finding Storage Hyperplanes Iteratively

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

each conflict is satisfied by at least one of the hyperplanes.

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

$$K_i' = K_i \cap \left\{ (\vec{s}, \vec{t}) \mid \vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0 \right\}, \quad 1 \leq i \leq l. \tag{3.7}$$

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

$$CS' = \cup_{i=1}^{i=l} K_i'. \tag{3.8}$$

In essence, instead of the original conflict set $CS$, the revised conflict set $CS'$ with its constituent conflict polyhedra $K_1', K_2', \ldots, K_l'$, forms the basis for determining the next storage hyperplane. If all the conflicts in a conflict polyhedron $K_i$ are satisfied by the hyperplane $\vec{\Gamma}$, its contribution to the revised conflict set $CS'$ due to the addition of the constraint $(\vec{\Gamma}.\vec{s} - \vec{\Gamma}.\vec{t} = 0)$ would be nothing. Therefore, this iterative process of determining storage hyperplanes is continued until all conflicts are satisfied, i.e., until the conflict set under consideration is empty. At each step, the contraction modulo is also found for every storage hyperplane.

Algorithm 1 summarizes the partitioning-based approach to find a modulo storage mapping. The main procedure, FIND-MODULO-MAPPING (line 1), determines the $m$ storage hyperplanes iteratively, revising the conflict set at each step as described above (lines 4-7). The procedure, FIND-NEXT-HYPERPLANE (line 10), sets up the ILP system (lines 12-17) necessary to determine the required storage hyperplane (line 18) and the corresponding contraction modulo.

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

1: **procedure** FIND-MODULO-MAPPING($A$, $CS$, $\vec{P}$)
2: $CS' \leftarrow CS$
3: $m \leftarrow 0$
4: **while** $CS' \neq \emptyset$ **do**
5: $m \leftarrow m + 1$
6: $(\Gamma_m, e_m) \leftarrow$ FIND-NEXT-HYPERPLANE($CS'$)
7: Revise the conflict set ($CS'$) as shown in (4.11) by revising the conflict polyhedra as shown in (3.7)
8: Let $M$ be the transformation matrix constructed with hyperplanes $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$ forming its rows
9: Let $\vec{e}$ be the vector of contraction moduli $e_1, e_2, \ldots, e_m$
**return** $(M, \vec{e})$
10: **procedure** FIND-NEXT-HYPERPLANE($CS'$)
11: $C \leftarrow \emptyset$
12: **for all** conflict polyhedra $K'_i \in CS'$ **do**
13: Formulate bounding constraints as shown in (3.4)
14: Formulate satisfaction decision constraints as shown in (3.5)
15: Apply Farkas' lemma to each of the above constraints (formulated in steps 13 and 14) to obtain an equivalent set of linear equalities/inequalities
16: Add the linear inequalities/equalities to $C$
17: Add the constraint on $\eta$ shown in (3.3) to $C$
18: Compute lexicographic minimal solution as shown in (3.6) to obtain the hyperplane $\Gamma$ and the corresponding contraction modulo $e$
**return** $(\Gamma, e)$

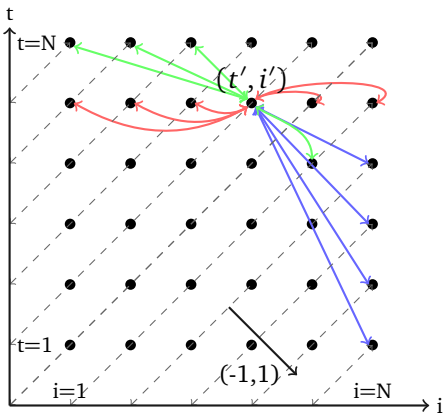



Figure 3.2: Storage hyperplane $(-1, 1)$ satisfies all conflicts

### 3.5.1 Example Revisited

Consider again the producer-consumer loops that were introduced in Figure 3.1. In Figure 3.2, conflicts in different polyhedra are shown in different colors. Note that the canonical hyperplanes $(1, 0)$ and $(0, 1)$ individually do not satisfy all conflicts — the former does not satisfy the conflicts colored in red whereas the latter does not satisfy those shown in blue. However, several other hyperplanes exist that satisfy all conflicts at once e.g. $(-1, 1)$, $(-2, 1)$, $(-3, 1)$ etc. Therefore, our greedy approach would pick such hyperplanes over other candidate hyperplanes. Furthermore, the secondary objective is to minimize the contraction modulo. Among such hyperplanes which satisfy all conflicts, $(-1, 1)$ leads to the smallest contraction modulo of $2N - 1$. Since all conflicts are satisfied by the hyperplane $(-1, 1)$ itself, there is no need to find any more partitioning hyperplanes. The resulting storage mapping, $A[t, i] \rightarrow A[(i - t) \bmod (2N - 1)]$, not only reduces the dimensionality but also provides a storage size requirement that is asymptotically better than that obtained using the successive modulo technique. This modulo storage mapping is also dimension and storage optimal.

### 3.5.2 Correctness and Termination

While the primary objective is to maximize conflict satisfaction for the revised conflict set, any hyperplane that is linearly dependent on the storage hyperplanes found in previous

iterations will not satisfy any new conflict. In practice, we observed that finding the next storage hyperplane using a revised conflict set is sufficient to ensure the required linearly independence of hyperplanes. If, in addition to revising the conflict set, a theoretical guarantee for such linear independence is sought, it can be enforced by introducing additional linear independence constraints, similar to those proposed in [BBK+08] for finding scheduling hyperplanes iteratively. Since the number of linearly independent storage hyperplanes required for satisfying all conflicts is at most equal to the dimensionality of the array space, the iterative process is guaranteed to terminate. A storage mapping that satisfies all conflicts is a valid one by definition: it maps conflicting indices to different partitions.

## 3.6 Optimality

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

1. In some pathological cases, a higher-dimensional mapping is better than a lower dimensional one, and this may not even be known at compile time. Consider a 2-d wavefront in a 3-d iteration space with a storage mapping of size $N_1 \times N_2$ versus a lower dimensional one with storage $N_3$. If $N_3 > N_1 \times N_2$, the higher-dimensional mapping leads to lower storage.

2. Since decision variables for conflict satisfaction are added on a per conflict polyhedron basis, splitting conflict polyhedra can only yield better solutions. This is also the case when splitting dependences or iteration domains leads to better parallelization.

3. Our first objective of conflict set satisfaction is greedy in nature and although not optimal, often finds optimal solutions in practice. The iterative approach to determine the partitioning hyperplanes (Section 3.5) is open to easy customization and

variation—for example to enumerate a fixed number of good solutions and pick the minimum storage one among them, given that our storage mapping determination is quite fast (Table 5.1).

Furthermore, if the primary objective of minimizing $\eta'$ in (3.6) results in it being equal to 0, it means that the storage hyperplane thus found can satisfy all conflicts by itself. The secondary objective ensures that the storage requirements of such a 1-dimensional modulo storage mapping found will be optimal.

## 3.7 Examples

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

### 3.7.1 Blur Filter - Interleaved Schedule

In image processing pipelines, such as Harris corner detectors [HS88] instead of time-iterated stencils, a pipeline stage may apply a particular stencil once, before propagating the computed output to the next stage, which then 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. [RKBA$^+$13] for their work on the Halide DSL compiler. Consider the loop nest of a 2-stage blur in Figure 3.3(a). The producer-consumer locality is quite poor. It can be improved by interleaving the horizontal and vertical blurs as shown in Figure 3.3(b). The values that are live out for the loop nest in Fig.3.3(b) are all stored in the array $out$ whereas $blurx$ only serves as an intermediate array necessary for computing the final live out values. As each statement instance $S(y,x)$ writes to $blurx[y,x]$ in accordance with the schedule $\theta(S(y,x)) = (y,x,0)$, the last use of the value $blurx[y,x]$ is in $T(y+2,x)$ at $\theta(T(y+2,x)) = (y+2,x,1)$. The conflict set $CS$ of the conflicting indices $(y,x) \bowtie (y',x')$ for the array space $blurx$ due to such a schedule is specified in Figure 3.4(a). The modulo storage mapping used by

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

(a) 2-stage blur filter

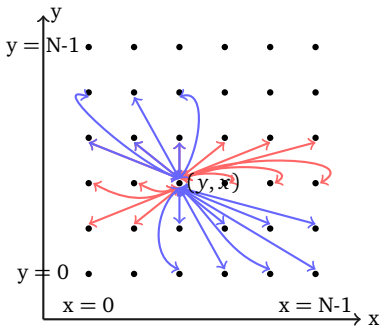
```
for (y=0; y<=N-1; ++y)
  for (x=0; x<=N-1; ++x) {
/*S*/ blurx[y,x] = in[x,y] + in[x+1,y] + in[x+2,y];
      if(y>=2)
/*T*/   out[x,y] = blurx[y-2,x] + blurx[y-1,x] + blurx[y,x];
   }
```

(b) Interleaved schedule

Figure 3.3: Different versions of 2-stage blur filter.

$$CS = \left((y,x),(y',x') \in blurx \right.$$
$$\wedge \left((y'-y \leq 1 \wedge y'-y \geq 0 \wedge x < x')\right.$$
$$\left.\left. \vee \ (y'-y \leq 2 \wedge y'-y \geq 1 \wedge x \geq x')\right)\right).$$

(a) Conflict set specification




(b) Note that $(y,x)$ does not conflict with the index $(y+2,x+1)$

Figure 3.4: Blur filter (interleaved schedule) – set of conflicts associated with index $(y,x)$ and their geometrical representation.

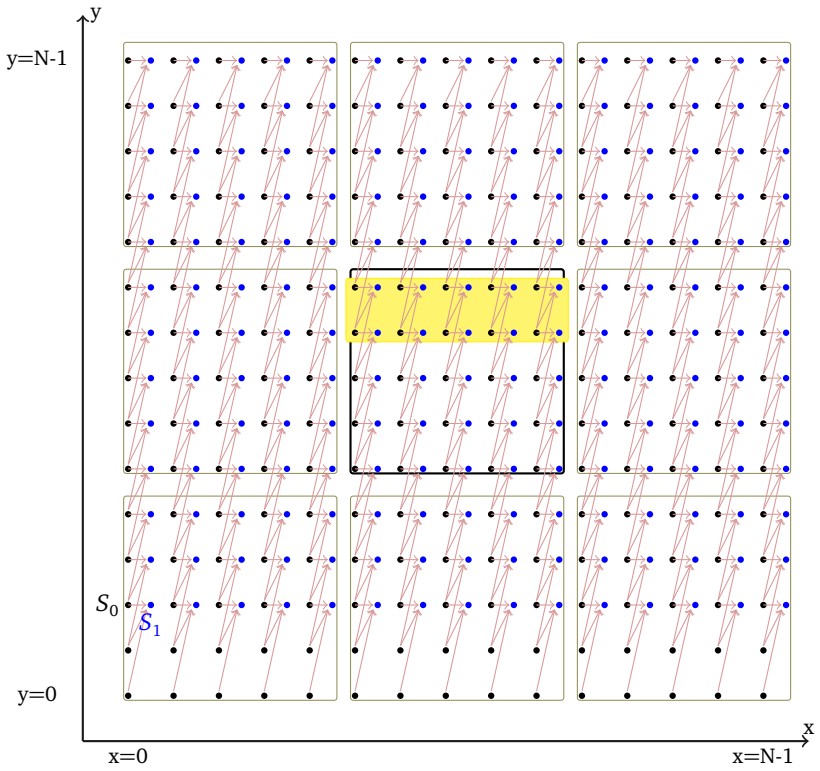
Ragan-Kelly et al. [RKBA+13] is same as that obtained using the successive modulo technique: $blurx[y,x] \rightarrow blurx[y \bmod 3, x \bmod N]$. Figure 3.4(b) shows that the storage hyperplane $(-1,2)$ would satisfy all the conflicts by itself. Furthermore, since it leads to the smallest contraction modulus of $2N+1$, the modulo storage mapping obtained using our technique is $blurx[y,x] \rightarrow blurx[(-y+2x) \bmod (2N+1)]$.

```
#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]
```

Figure 3.5: Tiled execution of blur filter: $tx$ and $ty$ are the tile iterators whereas $x$ and $y$ are the intra-tile iterators. $B$ is the tile size.

### 3.7.2 Blur filter - Tiled Execution

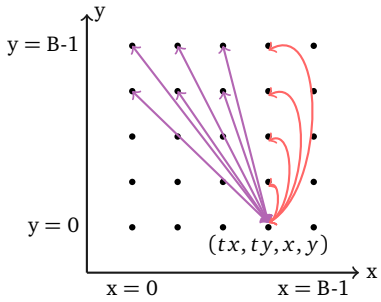
Figure 3.5 shows a tiled version of the blur filter code introduced in Figure 3.3(a). The schedules for the statements $S_0$ and $S_1$ can be expressed as $\theta(S_0(ty,tx,x,y)) = (ty,tx,x,0,y)$ and $\theta(S_1(ty,tx,x,y)) = (ty,tx,x,1,y)$. The column-wise processing is interleaved to further improve 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 3.6(a)). Prior to contraction, a total expansion of the array space written to by the statement $S_0$ is performed. This changes the write access to a 4-d access on an array space $A_0$, which has the same size and shape as the iteration domain of statement $S_0$ (refer Figure 3.5)). Now, consider the problem of contracting a data tile of the array space. The intra-tile conflict set, shown in Figure 3.6(b), specifies all the conflicts associated with index $(ty,tx,x,y)$ within the same data tile. The first disjunct in the conflict set is for indices conflicting with all other indices on the same column. The conflicts due to the live-out data values which are already computed are specified by the other disjunct. The storage mapping obtained using the successive modulo technique $A_0[ty,tx,x,y] \rightarrow A_0[ty,tx,x \bmod B, y \bmod B]$




(a) Live-out data for the tile in yellow. The black and blue dots represent instances of the statements $S_0$ and $S_1$ respectively.

$$CS = \left((x = x') \wedge (y' > y) \wedge (ty, tx, x, y), (ty, tx, x', y') \in A_0\right)$$
$$\vee \left((x' > x) \wedge (y \leq B-1) \wedge (y \geq B-2) \wedge (ty, tx, x, y), (ty, tx, x', y') \in A_0\right).$$

(b) Conflict set specification.



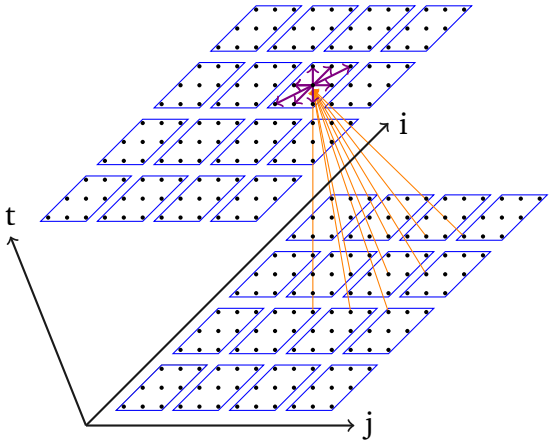

(c) A geometric representation of conflicts.

Figure 3.6: Blur filter (tiled execution) – conflict sets and their geometrical representation.

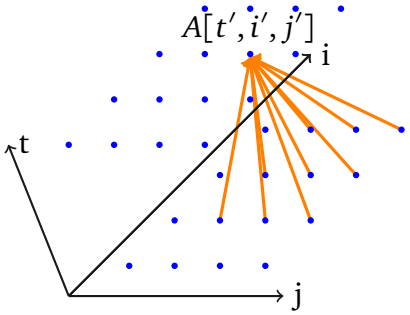
does not contract the tile at all. However, note that the hyperplane $(-2,1)$ satisfies all the conflicts with the contraction modulus $(3B-2)$. Our technique would arrive at this storage hyperplane to give the dimension and storage optimal storage mapping $A_0[ty,tx,x,y] \rightarrow A_0[ty,tx,(y-2x) \bmod (3B-2)]$. The storage required for the tile is thus reduced from $B^2$ down to $(3B-2)$. Also, the same storage mapping holds even if tiles along the same row are executed in parallel.

### 3.7.3 Lattice-Boltzmann Method (LBM)

We studied a discrete form of the Boltzmann equation [Suc01, PAVB15], used in computational fluid dynamics to model complex fluid flows. An LBM kernel is characterized as $DmQn$ where $m$ is the dimensionality of the space lattice and $n$ is the number of particle distribution equations that need to be solved. Figure 3.7(a) shows a $D2Q9$ lattice arrangement. Each blue box encapsulates the solutions of 9 particle distribution equations (the 9 black dots) for a particular particle being displaced through the 2-d space lattice. The neighborhood interactions in the $D2Q9$ example are such that if all the points in every blue box are collapsed into a single node representing all the associated computations, the flow dependences are similar to those of a typical stencil computation (refer Figure 3.7(b)). Suppose that the computations associated with a node $A[t,i,j]$ are performed at logical time $(t,i,j)$ and that the size of the array space is $N$ in each dimension. The conflict set $CS$ of conflicting indices $(t,i,j) \bowtie (t',i',j')$ for the array space $A$ is specified in 3.7(c). There is no hyperplane that satisfies all the conflicts on its own (refer Figure 3.7(d)). The canonical hyperplanes $(1,0,0)$, $(0,1,0)$,$(0,0,1)$ individually do not satisfy all conflicts either. Among the hyperplanes that satisfy all conflicts in three of the four conflict polyhedra, $(-2,1,0)$ leads to the smallest contraction modulus $(N+2)$. It satisfies all but the conflicts in violet. Since all the conflicts are not satisfied yet, another storage hyperplane must be found. The revised conflict set, containing only the unsatisfied conflicts, is essentially the conflict polyhedron made up of the violet conflicts. The hyperplane $(0,0,1)$ satisfies all of them with the smallest contraction modulus $N$. The final storage mapping obtained is



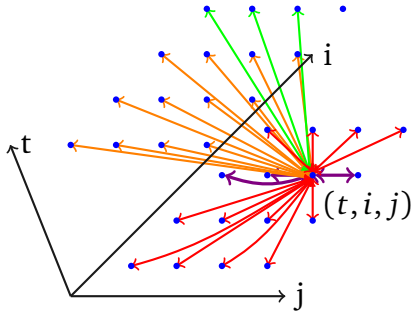

(a) $LBM - D2Q9$ in a pull model




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

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

(c) The conflict set specification.




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

Figure 3.7: As in a stencil, node $A[t',i',j']$, in Figure 3.7(b), depends on neighboring nodes from the previous time step.

$A[t,i,j] \rightarrow A[(i-2t) \bmod (N+2), j \bmod N]$ . The storage requirement of this 2-d mapping, $(N^2+2N)$ is marginally more than optimal storage size of $(N^2+N+1)$, i.e., the maximum number of live values at any point during the schedule.

### 3.7.4 Diamond Tiling

Loop tiling is a transformation which can be used to not only improve data locality but also to exploit coarse-grained parallelism. The shape of the tile and its size together characterize a loop tiling. However, with parallelization, a poor choice of the same can cause

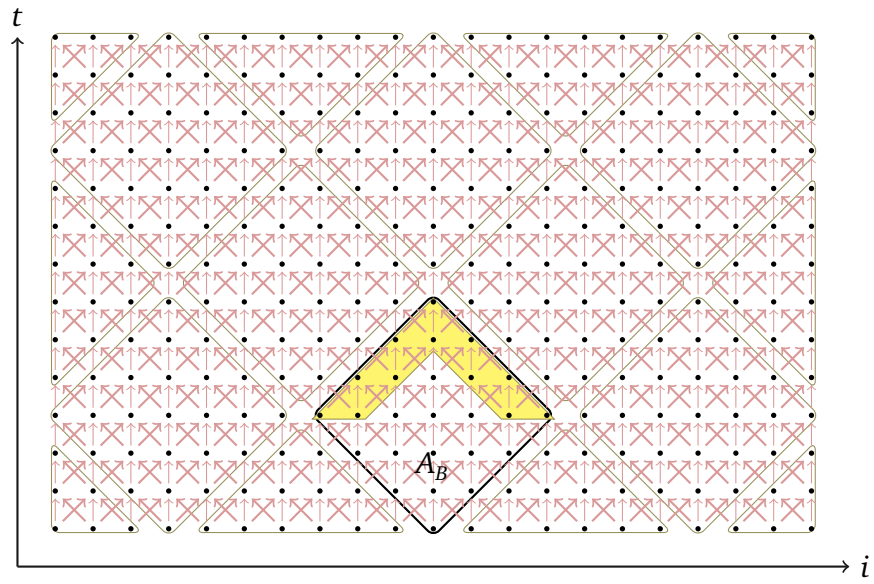



Figure 3.8: $A_B$ is the data tile written to by iterations within the tile outlined in black. Live-out data in yellow.

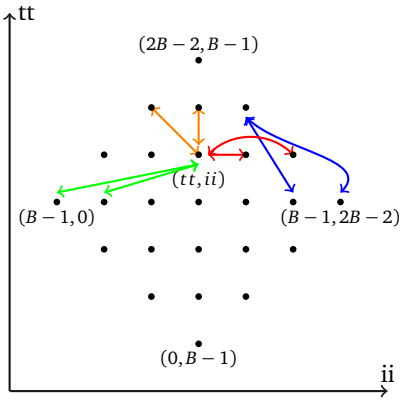
load imbalance due to pipelined start-up and drain delay. Bandishti et al. [BPB12] developed a technique to obtain a diamond tiling, thereby enabling the concurrent start of tiles. Figure 3.8 shows diamond tiling for a stencil with flow dependences $(1,-1)$, $(1,0)$ and $(1,1)$. If $B$ is the tile size, the live-out set is a union of polyhedra $L_1$ and $L_2$ (specified in Figure 3.9(a)). Let $(tt,ii)$ be the intra-tile iterators. Suppose that the intra-tile schedule is sequential so that the value written to the memory cell $A_B[tt,ii]$ is computed at time $(tt,ii)$. If so, the live-out portion of the data tile is as shown in Figure 3.8. The intra-tile conflicts, in accordance with the conflict set specification $CS$ of conflicting indices $(tt,ii) \bowtie (tt',ii')$, are as shown in Figure 3.9(b). Our algorithm selects the hyperplane $(1,-3)$ as it satisfies all conflicts by itself, resulting in contraction modulus $(6B-5)$. The final storage mapping $A_B[tt,ii] \rightarrow A_B[(tt-3ii) \bmod (6B-5)]$ is not only dimension-optimal but also has an asymptotically better storage requirement than that of $A_B[tt,ii] \rightarrow A_B[tt \bmod B, ii \bmod (2B-1)]$, which is found using successive modulo technique. This storage mapping would hold even if the tiles are executed in parallel.

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

(a) Live-out set for the diamond tile shown in Figure. 3.8

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

(b) Intra-tile conflict set specification. The last two disjuncts in the conflict set $CS$ represent conflicts due to flow dependences. The first two specify additional conflicts due to live-out values.




(c) All conflicts that are associated with the index $(tt, ii)$ shown above, can be expressed using just 3 polyhedra. The conflicts that are typically represented by the fourth conflict polyhedron in the conflict set specification are similar to those shown in blue.

Figure 3.9: Diamond tiling – conflict set and its geometric representation.

## 3.8 Enumerating Storage Mappings

The storage partitioning approach described in this chapter so far was published in [BBC16a], and as explained in Section 3.6, the solutions obtained using the partitioning approach summarized in Algorithm 1 need not always be optimal. The primary objective is to maximize conflict satisfaction by minimizing the number of conflict polyhedra that are left

unsatisfied and thereby, minimizing the dimensionality of the storage mapping. However, the storage requirements of higher dimensional mappings can often be less than that of lower dimensional mappings. So, it is possible that the storage mapping corresponding to the optimal storage requirement may be missed due to such an objective. In this section, we introduce a useful variation of the storage partitioning approach that enables the enumeration of various valid storage mappings instead of a single one. This facility provides the option of then picking the storage mapping with the least storage requirement among all those enumerated.

Consider again the example of diamond tiling from Section 3.7.4. The storage mapping obtained using the successive modulo technique and our intra-array storage optimization heuristic are $A_B[tt, ii] \rightarrow A_B[tt \bmod B, ii \bmod (2B-1)]$ and $A_B[tt, ii] \rightarrow A_B[(tt - 3ii) \bmod (6B-5)]$ respectively. The latter is dimension optimal and with a storage requirement of $6B-5$. It also provides an improvement in storage over the former. However, the optimal storage requirement for the diamond tile is $4B-2$; this corresponds to the modulo storage mapping $A_B[tt, ii] \rightarrow A_B[ii \bmod (2B-1), tt \bmod 2]$. This solution requires the partitioning of the array space using two storage hyperplanes, $(0,1)$ and $(1,0)$. Consequently, it is missed by our heuristic which is able to greedily satisfy all the conflicts using the single storage hyperplane $(1,-3)$.

### 3.8.1 Alternative Storage Hyperplanes

Suppose $\vec{\Gamma}$ is the storage hyperplane found by our greedy objective of conflict satisfaction for a given conflict set $CS = \cup_{i=1}^{i=l} K_i$. The general approach we take towards enumerating multiple valid storage mappings is to introduce constraints to our ILP formulation which facilitate the search for an alternative storage hyperplane $\vec{\Gamma}^{(I)}$. Such an alternative storage hyperplane is one which would have been discarded by our greedy objective in the absence of such constraints. Once such an alternative hyperplane is found, another alternative storage hyperplane, $\vec{\Gamma}^{(II)}$ can be found by constraining the ILP formulation further – and so on, until a certain criterion is met. In essence, the problem of enumerating storage mappings is turned into a problem of enumerating multiple storage hyperplanes.

Let $S$ be the set of conflict polyhedra satisfied by the storage hyperplane $\vec{\Gamma}$. Clearly, $|S| = \eta$. We experimented with several ways to constrain the search of alternative hyperplanes. The most effective way that we found was to constrain the search to the space of storage hyperplanes, all of which would satisfy a different set of conflict polyhedra. Consequently, the constraint for finding an alternative storage hyperplane can be specified as follows:

$$\Sigma_{K_i \in S} \left( x_{1i} + x_{2i} \right) < |S|. \tag{3.9}$$

Essentially, the required alternative storage hyperplane would satisfy a different set of conflict polyhedra if the sum of the decision variables associated with the conflict polyhedra satisfied by $\vec{\Gamma}$ is less than the number of conflict polyhedra satisfied by it. Note that this constraint does not exclude alternative storage hyperplanes which satisfy the same number of conflict polyhedra as $\vec{\Gamma}$, but a different set of them and potentially with a higher contraction modulus. A storage hyperplane $\vec{\Gamma}^{(II)}$ which is an alternative to both the storage hyperplanes $\vec{\Gamma}$ and $\vec{\Gamma}^{(I)}$ can then be found by adding constraints to ensure that it satisfies a set of conflict polyhedra which is different from that of both those hyperplanes. Alternative hyperplanes can be found in this manner until the ILP is constrained so much that the search space becomes empty – no subset of conflict polyhedra can possibly be satisfied with all the additional constraints for finding alternative storage hyperplanes. Each of the storage hyperplanes enumerated in this manner represents an alternative way of partitioning the given array space. As a result, each of them revises the conflict set $CS$ in its own way. Each revised conflict set can then serve as the basis for enumerating multiple storage hyperplanes at the next level.

Note that there could also be occasions when an alternative hyperplane $\vec{\Gamma}^{(I)}$ found in this manner is nothing but the same as the one already found. In this case, we need not consider this as an alternative solution. However, the ILP can be constrained further to obtain other alternatives. For example, consider a conflict set with three conflict polyhedra

$K_1, K_2, K_3$. Suppose the hyperplanes $\vec{h}$ and $\vec{h}'$ satisfy $K_1, K_2, K_3$ and $K_1, K_2$ respectively with corresponding contraction moduli $N$ and $2N$. Consequently, our heuristic will chose $\vec{h}$ over $\vec{h}'$. Now, if the ILP is solved again after adding the alternative storage hyperplane constraint (3.9), $\vec{h}$ will again be preferred over $\vec{h}'$ as it results in a smaller contraction modulus, even as it meets the new constraint for an alternative hyperplane by satisfying only two of the given three conflict polyhedra.

Algorithm 2 provides a high-level summary of the approach to enumerate various storage mappings for a given conflict set specification $CS$ associated with the array space $A$. Each element in the queue $Q$ captures the state of each alternative ongoing storage partitioning of the array space, with the pair – conflict set $CS$ and empty vector of hyperplanes – being enqueued to begin with (line 2). Each element in the queue is processed separately to find an alternative storage partition of the array space (line 3 to line 14). The procedure FIND-NEXT-HYPERPLANE is used to partition the array space by finding a hyperplane as in Algorithm 1 (since the only difference is that the ILP system set up for finding the next hyperplane is returned as well, the pseudo-code for it is not repeated here). The array space is iteratively partitioned, revising the conflict set at each iteration, until all the conflicts are satisfied (line 6). Furthermore, for each hyperplane found, possible alternatives to it are determined using the procedure FIND-ALTERNATIVE-HYPERPLANES (line 9). This procedure essentially adds the alternative storage hyperplane constraint, specified in (3.9), to the ILP system and solves it again using the same greedy objective of maximizing conflict satisfaction. If the alternative storage hyperplane found satisfies any conflict polyhedron in the conflict set i.e., either of the two decision variables associated with it is set to 1, the procedure is recursively called to determine other alternative hyperplanes — this proceeds until the ILP system is constrained so much that no hyperplane can be found that satisfies any conflict polyhedra. Furthermore, each of these partially complete alternative storage partitions are inserted into the queue for further partitioning (line 22).

The enumerative heuristic essentially finds alternative hyperplanes by constraining the search space so that conflicts in a different subset of conflict polyhedra are satisfied by each alternative storage hyperplane. Consequently, if $l$ is the number of conflict polyhedra

**Algorithm 2** Enumerate modulo storage mappings given a non-empty conflict set $CS$ for the array space $A$. $\vec{P}$ is the vector of program parameters.

1: **procedure** ENUMERATE-MODULO-MAPPINGS($A$, $CS$, $\vec{P}$)
2: Enqueue $(CS, ())$ to queue $Q$
3: **while** Q is not empty **do**
4: $(CS', \vec{H}) \leftarrow dequeue(Q)$
5: $m \leftarrow dim(\vec{H})$
6: **while** $CS' \neq \varnothing$ **do**
7: $m \leftarrow m + 1$
8: $((\Gamma_m, e_m), C) \leftarrow$ FIND-NEXT-HYPERPLANE($CS'$)
9: $Q \leftarrow$ FIND-ALTERNATIVE-HYPERPLANES($CS', C, Q, \vec{H}, \Gamma_m$)
10: Revise the conflict set $CS'$ (refer 4.11) by revising the conflict polyhedra as shown in (3.7)
11: $\vec{H} \leftarrow append(\vec{H}, (\Gamma_m, e_m))$
12: Let $M$ be the transformation matrix with hyperplanes $\Gamma_1, \Gamma_2, \ldots, \Gamma_m$ forming its rows
13: Let $\vec{e}$ be the vector of contraction moduli $(e_1, e_2, \ldots, e_m)$
14: Print the storage mapping characterized by : $(M, \vec{e})$
15: **procedure** FIND-ALTERNATIVE-HYPERPLANES($CS'$, $C$, $Q$, $\vec{H}$, $\Gamma$)
16: Add alternative hyperplane constraint for $\Gamma$ as shown in (3.9) to $C$
17: Compute lexicographic minimal solution as shown in (3.6) to obtain the hyperplane $\Gamma'$ and the corresponding contraction modulo $e'$
18: **if** $\Gamma'$ satisfied any conflict polyhedron in $CS'$
19: $Q \leftarrow$ FIND-ALTERNATIVE-HYPERPLANES($CS', C, Q, \vec{H}, \Gamma'$)
20: **if** $hyperplane\text{-}not\text{-}already\text{-}found(\vec{H}, \Gamma')$
21: Revise the conflict set $CS'$ (refer 4.11) by revising conflict polyhedra as shown in (3.7)
22: $Q \leftarrow enqueue(Q, (CS', append(\vec{H}, (\Gamma', e'))))$
**return** $Q$

in the input conflict set specification $CS$ for a given level of the enumerative heuristic, at most $2^l$ storage hyperplanes can be found at that level. This corresponds to the number of subsets of conflict polyhedra that are possible. With many of the techniques employed to determine storage hyperplanes having exponential worst-case complexity, this exponential number of possible alternative storage hyperplanes only worsens the complexity of the enumerative heuristic. However, in practice, we have seen that it is unlikely that a feasible storage hyperplane will be found for each possible subset of conflict polyhedra – the ILP constraints often drastically cut down the number of subsets which can be satisfied together. Furthermore, we have noticed that the quality of the storage mappings diminishes with the number of alternatives found. Therefore, a more practical approach would be to limit the number of alternatives to a small number. This can be considerably useful for array spaces of high dimensionality with potentially large number of conflict polyhedra.

### 3.8.2 Diamond Tiling Revisited

The intra-tile conflict set for a diamond tile, as shown in Fig. 3.9(b), is a union of four conflict polyhedra. Geometrically, conflicts in these four polyhedra are shown in Fig. 3.9(c) with double-headed arrows of the colours green, blue, red and orange respectively. As explained earlier, the storage hyperplane $(1,-3)$ satisfies all these conflicts on its own. Now, if Algorithm 2 is applied to enumerate various storage mappings, the first alternative storage hyperplane found is $(-1,0)$ which satisfies all conflicts except the red ones. Further search after constraining the ILP to ensure satisfaction of a different set of conflict polyhedra leads to the hyperplane $(0,-1)$ which satisfies all except the orange conflicts. Finally, a third alternative is found in the hyperplane $(-1,1)$ which does not satisfy the green conflicts but satisfies all others. At this point, the ILP has been constrained too much to obtain a different storage hyperplane that satisfies a different set of conflict polyhedra. Note that each of the four storage hyperplanes found so far satisfy a distinct set of conflict polyhedra. The last three of these i.e., $(-1,0)$, $(0,-1)$ and $(-1,1)$ each leave a single conflict polyhedron unsatisfied. These unsatisfied conflicts are then satisfied in the subsequent array space partitioning iteration through the storage hyperplanes $(0,1)$, $(1,0)$ and

$(0,1)$ respectively. Consequently, the four modulo storage mappings enumerated are as follows:

$$A[tt,ii] \rightarrow A[(tt-3ii) \bmod (6B-5)]$$
$$A[tt,ii] \rightarrow A[(-tt) \bmod B, ii \bmod (2B-1)]$$
$$A[tt,ii] \rightarrow A[(-ii) \bmod (2B-1), tt \bmod 2]$$
$$A[tt,ii] \rightarrow A[(-tt+ii) \bmod (2B-1), ii \bmod B]$$

Among these, it can be see that the third storage mapping has the least storage requirement of $(4B-2)$.

## 3.9 Related Work

Several array contraction techniques have been designed to reduce the memory footprint through a combination of dedicated loop transformations, mainly loop fusion and tiling, with a form of liveness analysis [MCT96, LLL01, Pik02]. In this work, we look for the most compact and efficient storage without questioning the quality of the loop nest's schedule or control flow. While early storage mapping optimizations [CDRV97, SCFS98] dealt with constant distance vectors, recent ones can operate on polyhedral abstractions [LF98, CC04, DSV05, ABD07]. The latter also benefit from array data-flow analysis to refine liveness information [DIY16b].

The intra-array optimization strategy of De Greef et al. [GCM97b] relies on the existence of a linearized schedule $\theta$, and on canonical linearizations of the array space. The reuse distance is computed as the maximum of the address differences between memory cells that are simultaneously live at any point during the entire execution schedule plus 1. A 1-d modulo storage mapping is obtained with the linearized access modulo the reuse distance. However, for the example in Figure 3.1, even with a linearized schedule of $\theta(t,i) = t*N+i$, it can be seen that this technique would not match the optimal solution

found using our technique. Clauss et al. [CFGV09] determine the storage requirement of affine loop nests by counting the points in a polyhedra and by maximization of polynomials. While polynomial mappings are more general, our work focuses on affine modulo mappings for which we could develop concrete cost functions using integer linear programming. Wilde and Rajopadhye [WR96], and later Quilleré and Rajopadhye [QR00], consider projective memory allocation functions to optimize memory usage in ALPHA programs. They introduced storage mapping optimization as the search for a low-dimensional linear projective allocation function. They also proposed an algorithm to minimize the dimensionality of the allocated arrays, but did not attempt to optimize for a more accurate model of the memory footprint, and did not consider scenarios where a portion of the variable is live-out, for example, in the case of tiled programs or the introductory example (Figure 1.1). As we have seen, this can introduce additional conflicts that do not arise due to flow dependences. Thus, for the benchmarks in Table 5.1 or Figure 1.1, their approach will be unable to improve mappings found by the successive modulo technique.

The notion of a conflict polyhedron was introduced by Darte et al. [DSV05, ABD07] in their work on lattice-based memory allocation. The bounds and heuristics explored by Darte et al. [DSV05] are under the assumption that the conflicting index difference set $DS$ is approximated as a 0-symmetric convex polyhedron. Our approach is fundamentally different—relying on the notion of *conflict satisfaction*, and works naturally with the conflict set expressed as a union of polyhedra. As we have seen, for tiled codes with boundary live-outs in multiple directions, our approach leads to an order of magnitude reduction in storage; this is a drastic reduction in memory, immediately observable in common practical cases. For the simple producer-consumer example in Figure 3.1, with $n = 9$, all heuristics implemented in Bee+cl@k [Ali07] determined the same modulo storage mapping of $a[t,i] \rightarrow a[t \bmod 9, i \bmod 9]$, using the same bases for computing the contraction moduli as that suggested by Lefebvre and Feautrier [LF98]. Their optimal search-based method for the convex approximation of the problem came up with the mapping, $a[t,i] \rightarrow a[t \bmod 1, (14t+i) \bmod 61]$, which clearly uses more storage than $a[t,i] \rightarrow a[(i-t) \bmod 17]$, obtained using our technique.

Following closely upon the heels of our work on intra-array storage reuse [BBC16a], Darte et al. have extended their lattice based allocation framework [DIY16a] to improve the quality of storage mappings obtained under it when the conflict constraints are a non-convex union of polyhedra. Furthermore, unlike the earlier lattice based heuristics that dealt with fixed basis, the generalized heuristics presented in their recent work are also extended with schemes to choose suitable bases. The basis selection scheme essentially minimizes the 'width' of the chosen direction at each step. Although this greedy approach can lead to sub-optimal solutions, it can easily be adapted to handle parametric constraints. Another heuristic, which complements the basis selection scheme, deals with reuse vectors instead of hyperplanes or directions of mapping. Any reuse vector found must be the integral vector with the smallest norm among all vectors that lie outside the extrusion of the non-convex union $K$, along the basis vectors which have already been found. Smaller the vector, greater will be the reuse. Furthermore, due to the difficulty in handling non-constant reuse vectors, parameters are handled by projecting them out in order to first find constant reuse vectors. Since this heuristic is not easily adapted to non-constant reuse vectors, a third combined heuristic is finally proposed with takes the best of both worlds: first, find as many constant reuse vectors as possible and then use the basis selection scheme to contract the array in the orthogonal space. The combined heuristic seems like a particularly clinical approach to the problem. However, as the authors themselves admit, it is not clear if the extended lattice based allocation schemes, which rely on reuse vectors, can be generalized to exploit inter-array reuse opportunities as we have done with the array partitioning approach (more on this generalization in Chapter 4).

Strout et al. [SCFS98] introduced the concept of an occupancy vector, which captures the duration (as a distance vector) after which a location can be reused in a repeated fashion. The universal occupancy vector (UOV) is one that is valid for any valid loop schedule, and a search-based approach is proposed to find the optimal UOV. The storage mappings that they derive from a UOV and our storage mappings are conceptually similar in that both specify an array partitioning, but differ in their mathematical form and in the approach used to find them. Since our mappings are for a particular schedule, they

are expected to lead to less storage: for example, assuming identity schedules, $(N+3)$ versus $2N$ for a 5-point 1-d stencil, $N^2+2N$ instead of $2N^2$ for LBM-D2Q9. However, schedule-independent UOV-based solutions for programs with constant dependences and multi-boundary live-outs (tiled or untiled) are still an order of magnitude better than those that use canonical bases [LF98], and those which are unable to find the right bases for such codes [QR00, DSV05, ABD07]. Another key difference is that the approach in [SCFS98] is designed for perfect loop nests with constant dependences. Thies et al. [TVSA01, TVA07] extend the notion of occupancy vector to an affine occupancy vector, which is valid for any valid affine schedule. They also discuss a technique for determining a storage mapping given an affine schedule. However, the given schedule needs to be one-dimensional thereby restricting the class of programs which lend themselves to their technique. In contrast, our technique supports multi-dimensional schedules even as it contracts the array along multiple dimensions.

The solution to the storage optimization problem discussed so far in this work pertains to intra-array reuse. The graph coloring technique prescribed for inter-array reuse in [LF98] is complementary to intra-array approaches, and can be used in conjunction to reduce the total number of arrays used in the program.

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

# CHAPTER 4

# AN INTEGRATED APPROACH TO STORAGE OPTIMIZATION

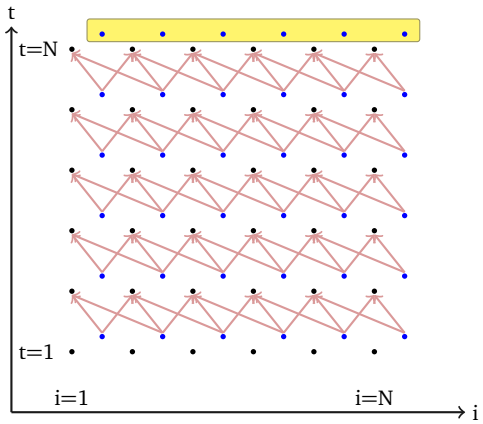
In the previous chapter, we discussed our approach based on array space partitioning for exploiting intra-array reuse opportunities. In order to exploit inter-array storage reuse opportunities, Lefebvre and Feautrier [LF98] proposed the construction of array interference graphs based on a straightforward computation of the rectangular hull. However, this can often fail to exploit the full potential for inter-statement storage reuse. In this chapter, 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.

## 4.1 A Simple Example

Consider the static control loop nest in Figure 1.2 introduced in Chapter 1, 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

```
1  // time-iterated stencil
2  for (t=1; t<=N; t++){
3    for (i=1; i<=N; i++)
4  /*S0*/ A0[t][i] = f((i>1 && t>1 ? A1[t-1][i-1] : Q[i-1]),
5                    (t>1 ? A1[t-1][i] : Q[i]),
6                    (i<N && t>1 ? A1[t-1][i+1] : Q[i+1]));
7    for (i=1; i<=N; i++)
8  /*S1*/ A1[t][i] = A0[t][i];
9  }
10 for(i=1; i<=N; i++)
11   result += A1[N][i];
```

(a) 1-d stencil from Figure 1.2 after total expansion.




(b) The instances of $S_1$ which compute the live-out data 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 Figure 4.1(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 Figure 4.1(a).

Figure 4.1: Inter-statement and intra-statement conflict sets for 1-d ping-pong style stencil 1.2.

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 Figure 4.1(a). A geometric representation of the iteration domains of statements $S_0$ and $S_1$ in Figure 1.2 is shown in Figure 4.1(b). The black and blue dots represent instances of

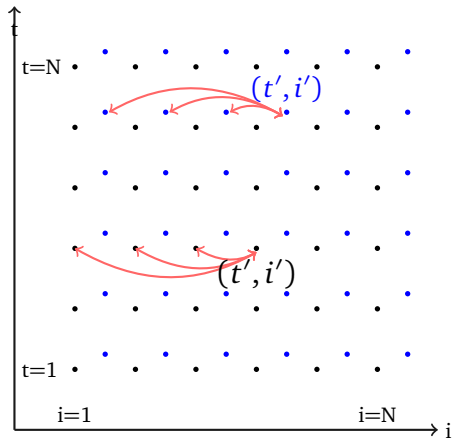



Figure 4.2: The red arrows denote the intra-statement conflicts (see Figure 4.1(c)).

the statements $S_0$ and $S_1$ respectively. The maroon arrows represent the flow dependences from $S_1$ to $S_0$.

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 Figure 4.1(c). Essentially, the top row computed by $S_1$ is live-out (refer Figure 4.1(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 Figure 4.1(c). Because 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 Figure 4.2. 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.

### 4.1.1 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

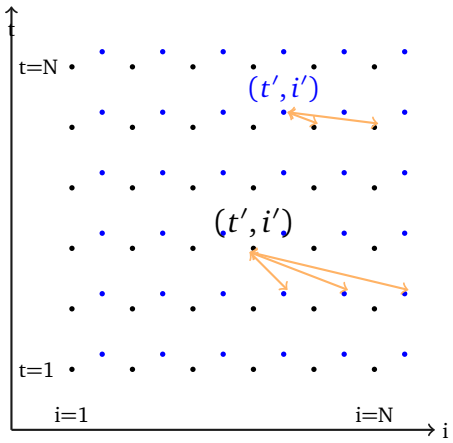



Figure 4.3: The orange arrows denote inter-statement conflicts (cf. Figure 4.1(d)).

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.

## 4.2 A Global Array Space

The process of total data expansion, described earlier in Section 2.4, 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.

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 Figure 4.1(a) are specified in Figure 4.1(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 Figure 4.3. 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 Figure 4.1(c) and Figure 4.1(d).

## 4.3 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 $\delta_s$ and $\delta_t$ 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 $\vec{\Gamma}_s$ and $\vec{\Gamma}_t$ 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 now characterize a hyperplane by both its normal and its offset (whereas so far, we only considered 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 Figure 4.1. 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 Figure 4.1(c) and Figure 4.1(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, consider the alternative storage mapping $A[j,t,i] \rightarrow A[(i-t) \bmod (N+1)]$. The overall storage requirement for such a mapping is $(N+1)$ which is less than that of the solution obtained using the canonical hyperplanes by a factor of two. Furthermore, by reducing the storage requirement further down to $(N+1)$, there is greater 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.

## 4.4 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 $\begin{pmatrix} \gamma_{l,1} & \gamma_{l,2} & \ldots & \gamma_{l,n} \end{pmatrix}$. The storage hyperplane offsets $\delta_j^{(1)}, \delta_j^{(2)}, \ldots, \delta_j^{(m)}$ make up the vector $\vec{\delta}_j$.

### 4.4.1 Global Conflict Set Specification

The global conflict set can be specified as a union of convex polyhedra. A few of these conflict polyhedra represent only the intra-statement conflicts, e.g. the conflict polyhedra in $CS_0$ and $CS_1$ (Figure 4.1(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 $CS_{1,0}$ (Figure 4.1(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.

## 4.5 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.

### 4.5.1 Analyzing Intra-Statement Conflicts

In this section, we generalize our approach from Chapter 3 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 $\delta_j$, 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 $\delta_j$ of the hyperplane is immaterial. This is also in accordance with the approach of Bhaskaracharya et al. [BBC16a], which was described in the previous chapter, 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 (3.4), we can formulate the following bounding constraint:

$$\begin{aligned} & \left(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}\right) \leq \left(\vec{u}_j.\vec{P} + w_j\right) \leq \left(c\vec{P} + c\right) \\ \wedge \quad & -\left(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}\right) \leq \left(\vec{u}_j.\vec{P} + w_j\right) \leq \left(c\vec{P} + c\right). \end{aligned} \tag{4.1}$$

Additionally, following the rationale behind the constraint (3.5), 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:

$$\begin{aligned} & \left(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t}\right) \geq 1 - \left(1 - x_{1i}\right)\left(c\vec{P} + c + 1\right) \\ \wedge \quad & \left(\vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_J.\vec{t}\right) \leq -1 + \left(1 - x_{2i}\right)\left(c\vec{P} + c + 1\right). \end{aligned} \tag{4.2}$$

In this way, each intra-statement conflict polyhedron 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 $\eta_{intra}$, all of whose conflicts are satisfied by the hyperplane found for the corresponding statement can then be estimated as follows:

$$\eta_{intra} = \sum_{\forall i,\ K_i \in CS_{intra}} \left(x_{1i} + x_{2i}\right). \tag{4.3}$$

In the previous chapter, we 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 $\eta_{intra}$ 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.

### 4.5.2 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 $\delta_j$ and $\delta_k$ 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:

$$\begin{aligned} &\left(\vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k\right) \leq \left(\vec{u'}_j.\vec{P}+w'_j\right) \leq \left(c\vec{P}+c\right) \\ \wedge \quad &-\left(\vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k\right) \leq \left(\vec{u'}_j.\vec{P}+w'_j\right) \leq \left(c\vec{P}+c\right). \end{aligned} \tag{4.4}$$

The inter-statement conflict difference could be positive, negative or equal to zero. Therefore, similar to the constraints in (4.2), 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 $K_i$:

$$\begin{aligned} &\left(\vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k\right) \geq 1-\left(1-x_{1i}\right)\left(c\vec{P}+c+1\right) \\ \wedge \quad &\left(\vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k\right) \leq -1+\left(1-x_{2i}\right)\left(c\vec{P}+c+1\right). \end{aligned} \tag{4.5}$$

The affine form of Farkas' lemma [Sch86, Fea92a] can be applied on the constraints formulated in (4.1), (4.4) and (4.2), (4.5), 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 $\eta_{inter}$, all of whose conflicts are satisfied by the hyperplanes found for the pair of statements associated with them can be estimated as follows:

$$\eta_{inter} = \sum_{\forall i,\ K_i \in CS_{inter}} \left(x_{1i}+x_{2i}\right). \tag{4.6}$$

### 4.5.3 A Greedy Objective

The resulting ILP system consists of constraints obtained due to the set of bounding constraints in (4.1) and (4.4), the decision constraints in (4.2) and (4.5) and also the constraints on $\eta_{intra}$ and $\eta_{inter}$ given by (4.3) and (4.6). 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 $\eta_{intra}$. 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 $(\vec{u}_j.\vec{P} + w_j)$ 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 [LF98], 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 $\vec{u}'_j = (u'^{(0)}, u'^{(1)}, \ldots, u'^{(\rho-1)})$, $\rho$ being the number of parameters involved, such a trade-off can be specified by the following constraint for each inter-statement conflict polyhedron $K_i$:

$$0 \leq \sum_{p=0}^{\rho-1} \left(u'^{(p)} - u^{(p)}\right) \leq \left(1 - x_{1i} - x_{2i}\right)(c\rho)\,. \tag{4.7}$$

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 $\eta_{inter}$ while minimizing the bound $(\vec{u'}_j.\vec{P}+w'_j)$ on the inter-statement conflict difference of every statement $S_j$.

$\eta_{intra}$ and $\eta_{inter}$ 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 $\eta_{intra}$, minimizing $(\vec{u}_j.\vec{P} + w_j)$, maximizing $\eta_{inter}$ 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:

$$\begin{aligned} minimize_{\prec} \Big\{ & \eta'_{intra}, u_0^{(0)}, u_0^{(1)}, \ldots, u_0^{(\rho-1)}, w_0, \ldots \\ & \ldots, u_{r-1}^{(0)}, u_{r-1}^{(1)}, \ldots, u_{r-1}^{(\rho-1)}, w_{r-1}, \\ & \eta'_{inter}, u'^{(0)}_0, u'^{(1)}_0, \ldots, u'^{(\rho-1)}_0, w'_0, \ldots \\ & \ldots, u'^{(0)}_{r-1}, u'^{(1)}_{r-1}, \ldots, u'^{(\rho-1)}_{r-1}, w'_{r-1} \}. \end{aligned} \tag{4.8}$$

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 $(\vec{u'}_j.\vec{P} + w'_j + 1)$.

### 4.5.4 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:

$$\forall K_i \in CS_{intra}, \quad K'_i = K_i \cap \left\{ (\vec{s}, \vec{t}) \mid \vec{\Gamma}_j.\vec{s} - \vec{\Gamma}_j.\vec{t} = 0 \right\}. \tag{4.9}$$

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:

$$\forall K_i \in CS_{inter}, K_i' = K_i \cap \Big\{ (\vec{s},\vec{t}) \mid \Big( \vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k = 0 \Big) \\ \vee \mid \vec{\Gamma}_j.\vec{s}+\delta_j-\vec{\Gamma}_k.\vec{t}-\delta_k \mid \geq min\Big(e_j,e_k\Big) \Big\}. \tag{4.10}$$

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

$$CS' = \cup_{1\leq i\leq l} K_i'. \tag{4.11}$$

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 3 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 4-7). The procedure, FIND-NEXT-HYPERPLANES (line 11), sets up the ILP system (lines 13-18) necessary to determine the required storage hyperplanes (line 19) and the corresponding

**Algorithm 3** 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.

1: **procedure** FIND-MODULO-MAPPINGS($A$, $CS$, $\vec{P}$)
2: $CS' \leftarrow CS$
3: $m \leftarrow 1$
4: **while** $CS' \neq \varnothing$ **do**
5: $(\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}, e_0^{(m)}, e_1^{(m)}, \ldots, e_{r-1}^{(m)}) \leftarrow$ FIND-NEXT-HYPERPLANES($CS'$)
6: Revise the conflict set $CS'$ using (4.11) by revising the conflict polyhedra using (4.9) and (4.10)
7: $m \leftarrow m + 1$
8: **for** $j \leftarrow 0$ to $r - 1$ **do**
9: 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
10: Let $\vec{e_j} = \left(e_j^{(1)}, e_j^{(2)}, \ldots, e_j^{(m)}\right)$, the vector of contraction moduli
**return** $(M_0, M_1, \ldots, M_{r-1}, \vec{e}_0, \vec{e}_1, \ldots, \vec{e}_{r-1})$
11: **procedure** FIND-NEXT-HYPERPLANES($CS'$)
12: $C \leftarrow \varnothing$
13: **for all** conflict polyhedra $K_i' \in CS'$ **do**
14: Formulate bounding constraints using (4.1) and (4.4)
15: Formulate satisfaction decision constraints using (4.2) and (4.5)
16: 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 $C$
17: Add constraint (4.7) for trading off inter-statement conflict satisfaction if $K_i \in CS_{inter}$
18: Add the constraint on $\eta_{intra}$ and $\eta_{inter}$ shown in (4.3) and (4.6) to $C$
19: Compute lexicographic minimal solution as shown in (4.8) 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** $(\Gamma_0, \Gamma_1, \ldots, \Gamma_{r-1}, e_0, e_1, \ldots, e_{r-1})$

contraction moduli.

### 4.5.5 Correctness, Termination and Optimality

The objective of successively partitioning the global array space is to ultimately satisfy all conflicts. As we observed earlier in Section 3.5.2, 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 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.

In the scenario when there is only one statement, note that our ILP formulation and our objective simplify to the one summarized in Algorithm 1 for the purpose of intra-array storage optimization. While our approach does not guarantee storage optimality in general, intra-array optimization is not penalized to allow inter-array reuse. This is due to the way we order the objective functions. As argued in Section 3.5.2, a claim on optimality cannot be made while not accounting for a number of orthogonal issues.

### 4.5.6 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 [LF98] 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 spaces 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 an $N \times 2$ 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 $(1, 0)$ respectively. Since neither of these two vectors is element-wise less than the other, the two arrays are better left uncoalesced. 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 low 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 Figure 4.1. In Figure 4.4, conflicts belonging to different conflict polyhedra (intra-statement as well as inter-statement) are shown in different colours. 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,

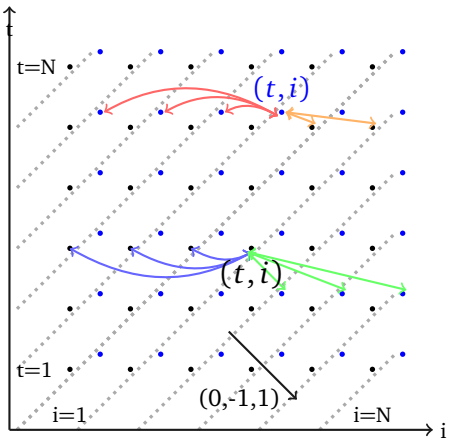



Figure 4.4: Storage hyperplane $(0,-1,1)$ satisfies all conflicts.

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 of two. In fact, the modulo storage mapping is storage optimal.

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 can thus be eliminated. Consequently, the loop enclosing statement $S_1$ can also be eliminated. The resulting code is a perfect loop nest with statement $S_0$.

## 4.6 Examples

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

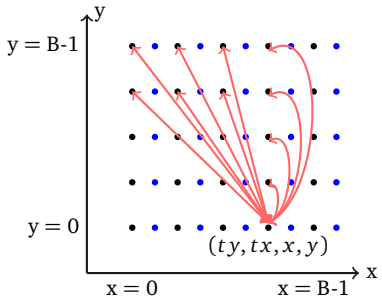
### 4.6.1 Blur filter

Consider again a tiled execution of the blur filter code shown in Figure 3.5, which was introduced earlier in Section 3.7.2. On total expansion, the write accesses in statements $S_0$ and $S_1$ are transformed to 4-dimensional accesses on the corresponding local array spaces $A_0$ and $A_1$, which have the same size and shape as their iteration domains. Suppose $A$ is the global unified array space such that $A[j] = A_j$ for $j = 0, 1$. Again, 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 $S_j$. The intra-tile conflict sets representing the conflicts $(j, x, y) \bowtie (j', x', y')$ are specified in Figure 4.5(a). $CS_0$ 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.

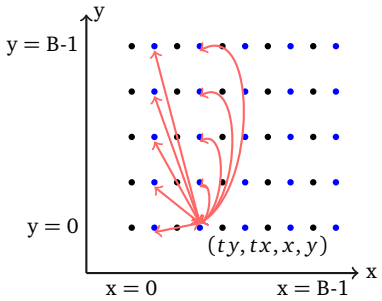
All the intra-statement conflicts of $S_0$ in $CS_0$ can be satisfied at once by the hyperplane $(0, 2, -1)$ with a maximum intra-statement conflict difference of $(3B - 3)$. However, since all values computed by $S_1$ are live-out, it can be seen that not all of the intra-statement conflicts of $S_1$ 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

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

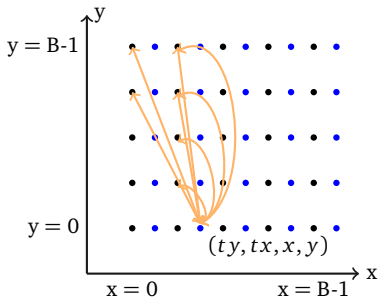
(a) The geometrical representations of these intra-tile conflict sets are shown below.



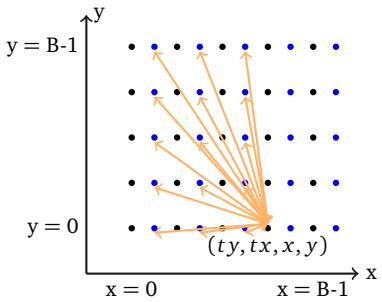

(b) Conflicts in $CS_0$




(c) Conflicts in $CS_1$




(d) Conflicts in $CS_{0,1}$




(e) Conflicts in $CS_{1,0}$

Figure 4.5: The conflict sets representing the intra-tile conflicts $(j, x, y) \bowtie (j', x', y')$ in the global array space $A$ are shown in Figure 4.5(a). Statements $S_0$ and $S_1$ write to the data tiles $A_{0T}$ and $A_{1T}$ respectively

$(B-1)$. 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 $S_1 : A_1[ty,tx,x,y] \rightarrow A'_1[ty,tx,x \bmod B, y \bmod N]$.

### 4.6.2 Smoothing

The geometric multi-grid algorithm [GV15] 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 [MVB15]. Figure 4.6 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 shown in Figure 4.7(d)). 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 Figure 4.7(a)—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 Figure 4.7(b), specifies that the index $(k,i,j)$ conflicts with all the 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 Figure 4.7(e). The former consists of the conflicts shown in red and violet, whereas the conflicts in orange and green represent the inter-statement conflicts.

```
#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];
```

Figure 4.6: Smoothing in multi-grid methods using the Jacobi 2-d stencil

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 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 Figure 4.7(c)), 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

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

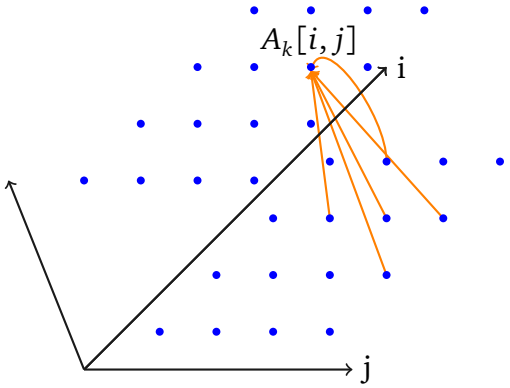
(a) The intra-statement conflict set specification for statement $S_k$ for $k = 0,1,\ldots,4$

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

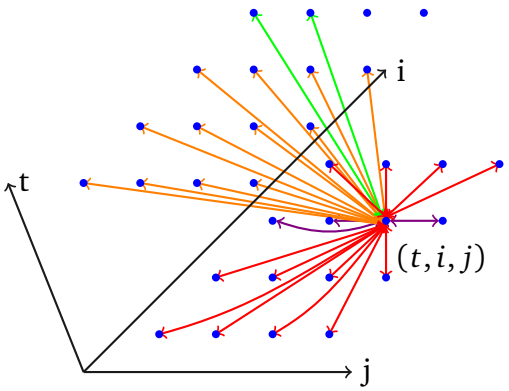
(b) The inter-statement conflict set specification for statements $S_k$ and $S_{k+1}$ for $k = 0,1,2,3$

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

(c) The global conflict set specification




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




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

Figure 4.7: Storage optimization of Jacobi 2-d smoothing in multi-grid methods

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 satisfy the red ones on

$$S_0 : A[0,i,j] \rightarrow A[(i+3) \mod (N+2)][j \mod N]$$
$$S_1 : A[1,i,j] \rightarrow A[(i+1) \mod (N+2)][j \mod N]$$
$$S_2 : A[2,i,j] \rightarrow A[(i-1) \mod (N+2)][j \mod N]$$
$$S_3 : A[3,i,j] \rightarrow A[(i-3) \mod (N+2)][j \mod N]$$
$$S_4 : A[4,i,j] \rightarrow A[(i-5) \mod (N+2)][j \mod N]$$

Figure 4.8: Storage mappings obtained for Jacobi 2-d smoothing (refer Figure 4.7)

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 Figure 4.8. 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.

## 4.7 Generalized Enumerative Heuristic

In Section 3.8, we described an approach for enumerating various modulo storage mappings to exploit intra-array reuse. The basic idea behind finding alternative storage mappings was to search for alternative storage hyperplanes by introducing suitable constraints. Similarly, if inter-array reuse opportunities are also factored in, it is possible to generalize this idea to enumerate various statement-wise storage mappings for a global array space by looking for alternatives to the set of statement-wise storage hyperplanes found. In other words, suppose $S$ is the set of conflict polyhedra satisfied by the statement-wise storage hyperplanes $\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}$ found at the $m^{th}$ level of storage partitioning using Algorithm 3. Clearly, $|S| = \eta_{intra} + \eta_{inter}$. An alternative collection of storage hyperplanes can

**Algorithm 4** Enumerate modulo storage mappings given a non-empty conflict set $CS$ for the array space $A$. $\vec{P}$ is the vector of program parameters.

**procedure** ENUMERATE-MODULO-MAPPINGS($A$, $CS$, $\vec{P}$)
  Enqueue $(CS, ())$ to queue $Q$
  **while** $Q \neq \emptyset$ **do**
    $(CS', \vec{H}) \leftarrow dequeue(Q)$
    $m \leftarrow dim(\vec{H})$
    **while** $CS' \neq \emptyset$ **do**
      $m \leftarrow m + 1$
      $((\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}, e_0^{(m)}, e_1^{(m)}, \ldots, e_{r-1}^{(m)}), C) \leftarrow$ FIND-NEXT-HYPERPLANES($CS'$)
      $Q \leftarrow$ FIND-ALTERNATIVE-HYPERPLANES($CS', C, Q, \vec{H}, \left(\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}\right)$)
      Revise the conflict set $CS'$ (refer 4.11) by revising the conflict polyhedra as shown in (3.7)
      $\vec{H} \leftarrow append(\vec{H}, (\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}, e_0^{(m)}, e_1^{(m)}, \ldots, e_{r-1}^{(m)}))$
    **for** $j \leftarrow 0$ to $r-1$ **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
    Print the storage mappings characterized by: $(M_0, M_1, \ldots, M_{r-1}, \vec{e}_0, \vec{e}_1, \ldots, \vec{e}_{r-1})$
**procedure** FIND-ALTERNATIVE-HYPERPLANES($CS'$, $C$, $Q$, $\vec{H}$, $\left(\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}\right)$)
  Add constraint for finding alternatives to hyperplanes $\left(\Gamma_0^{(m)}, \Gamma_1^{(m)}, \ldots, \Gamma_{r-1}^{(m)}\right)$ as shown in (4.12) to $C$
  Compute lexicographic minimal solution as shown in (4.8) to obtain the hyperplane $\left(\Gamma_0^{(m)'}, \Gamma_1^{(m)'}, \ldots, \Gamma_{r-1}^{(m)'}\right)$ and the corresponding contraction modulo $e_0^{(m)'}, e_1^{(m)'}, \ldots, e_{r-1}^{(m)'}$
  **if** a ny conflict polyhedron in $CS'$ satisfied by hyperplanes $\left(\Gamma_0^{(m)'}, \Gamma_1^{(m)'}, \ldots, \Gamma_{r-1}^{(m)'}\right)$
    $Q \leftarrow$ FIND-ALTERNATIVE-HYPERPLANES($CS', C, Q, \vec{H}, \left(\Gamma_0^{(m)'}, \Gamma_1^{(m)'}, \ldots, \Gamma_{r-1}^{(m)'}\right)$)
    **if** $same - hyperplanes\text{-}not\text{-}already\text{-}found(\vec{H}, \left(\Gamma_0^{(m)'}, \Gamma_1^{(m)'}, \ldots, \Gamma_{r-1}^{(m)'}\right)$)
      Revise the conflict set $CS'$ (refer 4.11) by revising conflict polyhedra as shown in (3.7)
      $Q \leftarrow enqueue(Q, (CS', append(\vec{H}, (\Gamma_0^{(m)'}, \Gamma_1^{(m)'}, \ldots, \Gamma_{r-1}^{(m)'}, e_0^{(m)'}, e_1^{(m)'}, \ldots, e_{r-1}^{(m)'}))))$
  **return** $Q$

then be found by introducing the following constraint:

$$\Sigma_{K_i \in S} \left( x_{1i} + x_{2i} \right) < |S|. \tag{4.12}$$

Notice that this is similar in form to the constraint (3.9). However, the set $S$ here includes both intra-statement as well as inter-statement conflict polyhedra that were satisfied by the hyperplanes found. A brief summary of the enumerative heuristic based on this generalization to the global array space is presented in Algorithm 4. The input to it is the global array space $A$ and the corresponding global conflict set $CS$. As in Algorithm 2, a queue $Q$ is used to keep track of each ongoing alternative storage partitioning of the global array space. For each collection of statement-wise storage hyperplanes found (line 8), alternatives to the same are found using the procedure FIND-ALTERNATIVE-HYPERPLANES (line 9). This procedure is similar to the procedure of the same name in Algorithm 2 — it is merely a generalized version which looks for alternatives to a collection of statement-wise hyperplanes instead of a single storage hyperplane. Consequently, we do not illustrate the behaviour of this algorithm with an example. Essentially, if only one statement is involved, the behaviour of Algorithm 4 would be similar to that of Algorithm 2.

## 4.8 Related Work

Most storage optimization techniques in the literature are intra-array ones. This includes those of Wilde and Rajopadhye [WR96], Lefebvre and Feautrier [LF98], Strout et al. [SCFS98], Quilleré and Rajopadhye [QR00], Thies et al. [TVSA01, TVA07], Darte et al. [DSV05, ABD07]. Among these, only Lefebvre and Feautrier [LF98] propose 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 1.2 presented motivation for such a unified approach.

In Chapter 3, we introduced the notion of storage hyperplanes and modeled the intra-array storage optimization as one of finding the right orientation for the storage hyperplanes. As seen in the examples that were discussed, this approach yields significant improvements in the quality of intra-array storage optimization over previous techniques. Our work described in this chapter builds on this approach, 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 [LF98] is decoupled from intra-array compaction; it thus misses mappings that can be obtained by taking a holistic view of all conflicts. De Greef et al. [GCM97b, GCM97a]'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 Figure 1.2.

# CHAPTER 5

# SMO - A POLYHEDRAL STORAGE OPTIMIZER

In this chapter, we present the details of the experimental evaluation of the storage optimization techniques described in the previous chapters. We implemented these storage optimization heuristics based on the array partitioning approach into an automatic storage optimizer, SMO [SMO16], using ISL [Ver10] (version isl 0.12.2) with GLPK (GNU Linear Programming kit) [GNU10] version 4.45 as the ILP solver. SMO is open source and public available for download. The input to SMO is 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. In the scenario when only one statement is involved, the global conflict set specification defines the set of conflicts associated with the array space written by the statement.

| Benchmark | | Modulo storage mapping | Reduction (approx.) | SMO time |
|---|---|---|---|---|
| produce-consume (Fig.3.1) | baseline | $A[t \bmod N, i \bmod N]$ | $\frac{N}{2}$ | |
| | SMO | $A[(i-t) \bmod (2N-1)]$ | | 0.17s |
| blur-interleaved (Fig.3.3(b)) | baseline | $blurx[y \bmod 3, x \bmod N]$ | 1.5 | |
| | SMO | $blurx[(2x-y) \bmod (2N+1)]$ | | 0.14s |
| blur-tiled (Fig.3.6) | baseline | $A[ty, tx, x \bmod B, y \bmod B]$ | $\frac{B}{3}$ | |
| | SMO | $A[ty, tx, (y-2x) \bmod (3B-2)]$ | | 0.11s |
| harris-corner-tiled | baseline | $sobel[tx, ty, x \bmod B, y \bmod B]$ | $\frac{B}{3}$ | |
| | SMO | $sobel[tx, ty, (y-2x) \bmod (3B-2)]$ | | 0.12s |
| unsharp-mask-tiled | baseline | $A[z, tx, ty, x \bmod B, y \bmod B]$ | $\frac{B}{5}$ | |
| | SMO | $A[z, tx, ty, (y-4x) \bmod (5B-4)]$ | | 0.82s |
| LBM-D2Q9 (Fig.3.7) | baseline | $A[t \bmod 2, i \bmod N, j \bmod N]$ | 2 | |
| | SMO | $A[(i-2t) \bmod (N+2), j \bmod N]$ | | 0.61s |
| LBM-D3Q19 | baseline | $A[t \bmod 2, i \bmod N, j \bmod N, k \bmod N]$ | 2 | |
| | SMO | $A[(i-2t) \bmod (N+2), j \bmod N, k \bmod N]$ | | 3.32s |
| LBM-D3Q27 | baseline | $A[t \bmod 2, i \bmod N, j \bmod N, k \bmod N]$ | 2 | |
| | SMO | $A[(i-2t) \bmod (N+2), j \bmod N, k \bmod N]$ | | 3.33s |
| diamond-tile (Fig.3.9) | baseline | $A_B[tt \bmod B, ii \bmod (2B-1)]$ | $\frac{B}{3}$ | |
| | SMO | $A_B[(tt-3ii) \bmod (6B-5)]$ | | 0.44s |
| stencil-1d-pllgm-tile | baseline | $A_B[tt \bmod B, ii \bmod B]$ | $\frac{B}{3}$ | |
| | SMO | $A_B[(tt-ii) \bmod (3B-2)]$ | | 0.29s |
| stencil-1d-hex-tile | baseline | $A_B[tt \bmod B, ii \bmod (3B-2)]$ | $\frac{B}{3}$ | |
| | SMO | $A_B[(-tt+3ii) \bmod (9B-8)]$ | | 1.15s |

Table 5.1: Storage reduction obtained using our approach (SMO) compared to the baseline successive modulo technique ( [LF98]) with $B$ being the loop blocking factor

| Benchmark | Input size | Execution time | | Speedup | Storage |
|---|---|---|---|---|---|
| | | baseline | smo | | reduction |
| blur-interleaved (Fig.3.3(b)) | 8192×8192 | 1.280s | 1.815s | 0.705× | 1.50× |
| blur-tiled (Fig.3.6) | 8192×8192, B=512 | 0.046s | 0.033s | 1.389× | 170.8× |
| unsharp-mask-tiled | 4096×4096, B=512 | 0.674s | 0.602s | 1.120× | 102.6× |
| harris-corner-tiled | 8192×8192, B=64 | 0.716s | 0.604s | 1.185× | 21.56× |
| LBM-D2Q9 (Fig.3.7) | 1024×1024, T=500 | 14.93s | 18.11s | 0.824× | 2.00× |
| LBM-D3Q19 | 200×200×200, T=100 | 79.21s | 83.62s | 0.947× | 2.00× |
| LBM-D3Q27 | 200×200×200, T=100 | 113.8s | 132.1s | 0.861× | 2.00× |
| diamond-tile (Fig. 3.9) | N=T=8192, B=256 | 1.489s | 1.506s | 0.988× | 85.44× |
| stencil-1d-pllgm-tile | N=T=8192, B=8 | 1.584s | 1.617s | 0.979× | 2.91× |

Table 5.2: Performance of various benchmarks with the storage mappings shown in Table 5.1

| Benchmark | | Speedup | Demand data L2 miss rate | Demand data L3 miss rate | % cycles in memory access (LLC misses) | % cycles in LLC access (L2 misses that hit in LLC) |
|---|---|---|---|---|---|---|
| blur-interleaved (Fig.3.3(b)) | baseline | 0.705× | 0.0569 | 0 | 0 | 0.0358 |
| | smo | | 0.0043 | 0 | 0 | 0.0031 |
| blur-tiled (Fig.3.6) | baseline | 1.389× | 0.1250 | 0 | 0 | 0.0031 |
| | smo | | 0 | 0 | 0 | 0 |
| unsharp-mask-tiled | baseline | 1.120× | 0 | 0 | 0 | 0 |
| | smo | | 0 | 0 | 0 | 0 |
| harris-corner-tiled | baseline | 1.185× | 0.3975 | 0 | 0 | 0.2351 |
| | smo | | 0.4284 | 0 | 0 | 0.3509 |
| LBM-D2Q9 (Fig.3.7) | baseline | 0.824× | 0.0219 | 0 | 0 | 0.0288 |
| | smo | | 0.0083 | 0 | 0 | 0.0089 |
| LBM-D3Q19 | baseline | 0.947× | 0.0366 | 0.0309 | 0.0084 | 0.0507 |
| | smo | | 0.0412 | 0.0478 | 0.0137 | 0.0520 |
| LBM-D3Q27 | baseline | 0.861× | 0.1390 | 0.0099 | 0.0097 | 0.1850 |
| | smo | | 0.1643 | 0.0107 | 0.0104 | 0.1828 |
| diamond-tile (Fig. 3.9) | baseline | 0.988× | 0.1763 | 0 | 0 | 0.0091 |
| | smo | | 0.1673 | 0 | 0 | 0.0083 |
| stencil-1d-pllgm-tile | baseline | 0.979× | 0.1953 | 0 | 0 | 0.0037 |
| | smo | | 0.2188 | 0 | 0 | 0.0040 |

Table 5.3: Analysis of the performance of various benchmarks (shown in Table 5.2) using VTune

| Benchmark | | Speedup | DTLB overhead | % pipeline slots retired per cycle | CPI | LEA stalls |
|---|---|---|---|---|---|---|
| blur-interleaved (Fig.3.3(b)) | baseline | 0.705× | 0.0159 | 0.0289 | 11.970 | 0 |
| | smo | | 0.0062 | 0.0813 | 3.8828 | 0 |
| blur-tiled (Fig.3.6) | baseline | 1.389× | 0 | 0.1816 | 1.4876 | 0.0120 |
| | smo | | 0 | 0.2641 | 1.0248 | 0.0161 |
| unsharp-mask-tiled | baseline | 1.120× | 0 | 0.2209 | 1.4214 | 0 |
| | smo | | 0.0004 | 0.2279 | 1.3727 | 0 |
| harris-corner-tiled | baseline | 1.185× | 0.0008 | 0.2501 | 1.2204 | 0.0144 |
| | smo | | 0.0004 | 0.2724 | 1.1218 | 0.0054 |
| LBM-D2Q9 (Fig.3.7) | baseline | 0.824× | 0.0004 | 0.3943 | 0.6551 | 0 |
| | smo | | 0 | 0.4641 | 0.5271 | 0.0457 |
| LBM-D3Q19 | baseline | 0.947× | 0 | 0.2726 | 0.9526 | 0.0000 |
| | smo | | 0 | 0.3447 | 0.7261 | 0.0311 |
| LBM-D3Q27 | baseline | 0.861× | 0.0003 | 0.3274 | 0.7931 | 0.0000 |
| | smo | | 0 | 0.3523 | 0.7168 | 0.0286 |
| diamond-tile (Fig. 3.9) | baseline | 0.988× | 0.0021 | 0.2087 | 1.4699 | 0.0027 |
| | smo | | 0.0036 | 0.2191 | 1.4056 | 0.0051 |
| stencil-1d-pllgm-tile | baseline | 0.979× | 0 | 0.1110 | 2.3634 | 0.0060 |
| | smo | | 0.0000 | 0.1395 | 1.9144 | 0.0108 |

Table 5.4: Analysis of the performance of various benchmarks (shown in Table 5.2) using VTune (continued from Table 5.3)

## 5.1 Storage Mappings for Contracting Intra-Array Storage

Table 5.1 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 stencil benchmarks were optimized for cache locality using the Pluto heuristic [Plu]. The unsharp-mask and harris-corner kernels were taken from PolyMage [MVB15] while the LBM benchmarks are due to the work of Pananilath et al. [PAVB15]. Note that in all cases where we perform tiling, tile sizes are fixed at compile time - the factor $B$ in Table 5.1 and elsewhere is only to clarify the relationship between storage reduction and tile size.

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

Suppose the tiling hyperplanes for the stencil in Fig. 3.9 were $(1,0)$ and $(1,1)$. Intra-tile storage optimization for such a parallelogram shaped tile with pipelined start-up would yield the storage mapping $A_B[tt, ii] \rightarrow A_B[(tt - ii) \bmod (3B-2)$. Similarly, for a hexagonal tile [GCH+14], the storage mapping obtained is $A_B[tt, ii] \rightarrow A_B[(-tt+3*ii) \bmod (9B-8)$.

**Access Expression Simplification** The form of our mapping is the same as in any other successive modulo optimization technique—so we do not introduce any more modulo expressions than previous ones. In fact, since our technique reduces the dimensionality of the storage mapping better than previous techniques (for example, blur and Harris corner detection benchmarks), we will have fewer modulo expressions. Note that any potential slowdown due to the modulo expression is avoided in several cases due to the finite bounds on the affine accesses. If an access expression $(y - 2x)$ ranges from say, $-2B + 2$ to $B - 1$, subtracting a base function from the access eliminates the modulus, e.g. $A[(y - 2x) \bmod 3B - 2]$ can be converted to $A[(y - 2x + 2B - 2)]$. Additionally, if

the modulus is a power of two or just less than it, the modulo expression can be replaced with a mere bit-wise left shift. The additional arithmetic operations introduced due to the optimized storage mappings are simple integer ones. Many of the expressions are also invariant with respect to the innermost loop. Such integer arithmetic is well hidden in the pipeline, which is good for performance, but obfuscates performance analysis (its effects in isolation cannot be accurately characterized).

### 5.1.1 Impact on Performance and Analysis

Table 5.2 gives the execution times of various benchmarks observed when the storage mappings shown in Table 5.1 were used. For the benchmarks blur-tiled, harris-corner-tiled, diamond-tile, stencil-1d-pllgm-tile, the tiles were executed in parallel using OpenMP. All the benchmarks were compiled with Intel C compiler (version 15.0) with flags "-O3 -openmp" and run on all cores of an Intel Xeon E5-2680 dual-socket machine with 8 cores per socket and a total of 64 GB of non-ECC RAM. The execution times were reproducible (less than 4% variation). It can be seen that for several benchmarks, the execution times with our storage mappings were the same if not better than those with mappings obtained using the successive modulo technique. We also observed that a high reduction factor in storage (tens of times) does not necessarily translate to a high performance gain. This is explained by the fact that in case of the tiled benchmarks we use a locality optimized code as the starting point for memory optimization; the benchmarks stencil-tile and stencil-1d-pllgm-tile were optimized for locality using the Pluto algorithm. Furthermore, it is not surprising that a big reduction in memory footprint does not result in a performance improvement in certain cases. If the code is tiled for the L2 cache, storage optimization alone may not further reduce stalls due to loads and stores (as cache miss rates would have already been low). As an analogy, reducing the memory footprint say from 40 GB to 400 MB, an application may still remain compute or memory bandwidth-bound as the case may be. On the other hand, the application workload will now scale 100× with respect to data on the same hardware.

To better understand the performance implications of storage optimization using a

modulo mapping, we analyzed all benchmarks using Intel VTune [Int15]. The only difference between the baseline and smo version of the benchmarks was in the array access expressions, and array definitions reflecting the reduced storage use. A summary of the profiling results is presented in Table 5.3 and Table 5.4. The demand data L2 miss rate is computed as the ratio of the sum of all types of L2 demand data misses to the sum of L2 demand data requests; similarly, for the demand data L3 miss rate. LEA stalls are determined as the ratio of the number of cycles with at least one slow LEA (load effective address) micro-operation being allocated to the number of core cycles when the core is not in halt state. Zero entries typically stand for a negligible value for the specific metric—due to sampling used by the profiler. Formulae for all other profiling metrics can be found in the VTune guide for Intel Xeon Processor E5 family [Int13].

The three benchmarks showing a clear speedup due to storage optimization are blur-tiled, unsharp-mask-tiled and harris-corner-tiled. Among these, there is a decrease in demand data L2 miss rate due to storage optimization only for blur-tiled. In case of the harris-corner-tiled benchmark, the demand data L2 miss rate remains the same, but the LEA stalls decrease. On the other hand, there is a definite increase in LEA stalls for all LBM benchmarks contributing to the overall slowdown (close to 15% in case of LBMD2Q9 and LBMD3Q27). The impact of SMO on access expressions complexity can be seen on benchmarks with 3-d arrays, and this is reflected in LEA stalls. While L2 and L3 miss rates remain nearly the same for LBMD3Q19 and LBMD3Q27, there is actually a decrease in the L2 miss rate for LBMD2Q9 due to storage optimization. Overall, the CPI metric improves (decreases) for all the benchmarks due to storage optimization, although the impact of SMO on instruction count may sometimes be significant. The blur-interleaved benchmark shows one of the most drastic reductions in CPI, from 11.97 to 3.88, while a 0.705× slowdown is incurred by this benchmark. This counter-intuitive result can be attributed to the code size increase (implied by the big change in CPI) given that all profiling metrics (L2 miss rate, DTLB overhead) show some improvement due to storage optimization.

| #Processes | N=1024,T=10 | | | N=8192,T=10 | | | N=12000,T=10 | | |
|---|---|---|---|---|---|---|---|---|---|
| | baseline | smo | Speedup | baseline | smo | Speedup | baseline | smo | Speedup |
| 3 | 0.3s | 0.368s | 0.814 × | 19.94s | 23.47s | 0.849 × | 41.26s | 52.58s | 0.784 × |
| 5 | 0.333s | 0.383s | 0.870 × | 20.07s | 24.62s | 0.815 × | 45.59s | 54.86s | 0.830 × |
| 6 | 0.321s | 0.388s | 0.826 × | 19.94s | 23.91s | 0.833 × | 552.1s | 53.44s | 10.32 × |
| 7 | 0.346s | 0.402s | 0.861 × | 21.55s | 26.61s | 0.810 × | 11253s | 444.7s | 25.30 × |
| 9 | 0.390s | 0.412s | 0.947 × | 23.47s | 25.88s | 0.907 × | | | |
| 11 | 0.387s | 0.408s | 0.95 × | 25.77s | 26.67s | 0.966 × | | | |
| 13 | 0.451s | 0.432s | 1.045 × | 350.4s | 27.87s | 12.57 × | | | |
| 14 | 0.444s | 0.428s | 1.036 × | 1226.9s | 102.2s | 12.00 × | | | |
| 15 | 0.453s | 0.427s | 1.061 × | 12974.9s | 92.46s | 140.3 × | | | |
| 16 | 0.457s | 0.434s | 1.053 × | | | | | | |

Table 5.5: Execution time of multiple instances of LBMD2Q9 being run in a multiprogrammed fashion.

| #Processes | N=200,T=10 | | | N=275,T=10 | | | N=350,T=10 | | |
|---|---|---|---|---|---|---|---|---|---|
| | baseline | smo | Speedup | baseline | smo | Speedup | baseline | smo | Speedup |
| 3 | 12.36s | 14.00s | 0.883 × | 31.34s | 29.71s | 1.054 × | 61.96s | 75.66s | 0.818 × |
| 5 | 13.27s | 14.73s | 0.901 × | 34.49s | 33.67s | 1.024 × | 71.44s | 78.40s | 0.911 × |
| 6 | 12.91s | 14.58s | 0.885 × | 33.37s | 32.64s | 1.022 × | 66.64s | 75.81s | 0.879 × |
| 7 | 14.52s | 16.06s | 0.904 × | 35.87s | 36.66s | 0.978 × | 808.4s | 172.8s | 4.677 × |
| 8 | 13.75s | 15.41s | 0.891 × | 35.27s | 34.88s | 1.011 × | 5841.3s | 1511.7s | 3.864 × |
| 9 | 15.25s | 16.40s | 0.929 × | 39.28s | 37.12s | 1.058 × | | | |
| 11 | 16.95s | 18.23s | 0.929 × | 42.87s | 42.28s | 1.013 × | | | |
| 13 | 18.43s | 19.16s | 0.961 × | 92.05s | 45.55s | 2.020 × | | | |
| 15 | 20.29s | 21.03s | 0.964 × | 1570.7s | 148.2s | 10.59 × | | | |
| 16 | 20.01s | 20.61s | 0.970 × | | | | | | |

Table 5.6: Execution time of multiple instances of LBMD3Q27 being run in a multiprogrammed fashion.

**Multiprogramming** Reduction in storage requirement can also potentially increase the degree of multiprogramming due to a reduction in virtual memory swapping through greater utilization of resident memory. We ran multiple instances of the LBMD2Q9 and LBMD3Q27 benchmarks in parallel in order to analyze this impact on multiprogramming. The time taken for running a different number of instances of each of them for three different input sizes are shown in Table 5.5 and Table 5.6. The performance trends clearly demonstrate that as the number of benchmark instances increases, the performance of multiple instances of a storage optimized version continues to match that of running multiple instances of the corresponding baseline version. On the contrary, the performance drops sharply for the baseline version when running more instances. Finally, when the number of processes becomes large enough for disk access to become a dominant factor in the execution time, we see very high speedups with the storage optimized versions over corresponding baseline ones.

## 5.2 Storage Mappings for Exploiting Inter-Array Reuse

Table 5.8 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 in a ping-pong fashion as shown in Figure 1.2), tiled versions of blur filter and unsharp mask image processing kernels [MVB15], and Jacobi smoothing iterations used in Multigrid methods [GV15]. The overall storage was approximately reduced by a factor of two 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. 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 5.7 compares execution time of the benchmarks

| 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 5.7: Benchmark performance with the storage mappings of Table 5.8

| Benchmark | Modulo storage mapping | | Reduction | SMO |
|---|---|---|---|---|
| | baseline | SMO | (approx.) | time |
| 1-d stencil (Fig.4.1) | $S_0 : A_0[t \bmod 1, i \bmod N]$<br>$S_1 : A_1[t \bmod 1, i \bmod N]$ | $A[(i-t) \bmod (N+1)]$<br>$A[(i-t) \bmod (N+1)]$ | 2 | 0.055s |
| 2-d stencil | $S_0 : A_0[t \bmod 1, i \bmod N, j \bmod N]$<br>$S_1 : A_1[t \bmod 1, i \bmod N, j \bmod N]$ | $A[(i-3t+1) \bmod (N+2), j \bmod N]$<br>$A[(i-3t) \bmod (N+2), j \bmod N]$ | 2 | 0.633s |
| 3-d stencil | $S_0 : A_0[t \bmod 1, i \bmod N, j \bmod N, k \bmod N]$<br>$S_1 : A_1[t \bmod 1, i \bmod N, j \bmod N, k \bmod N]$ | $A[(i-3t) \bmod (N+2), j \bmod N, k \bmod N]$<br>$A[(i-3t-1) \bmod (N+2), j \bmod N, k \bmod N]$ | 2 | 22.57s |
| jacobi-2d-smoothing (Fig.4.6) | $S_k : A_{k\%2}[i \bmod N, j \bmod N]$ | $A[(i+3-2k) \bmod (N+2), j \bmod N]$ | 2 | 4.846s |
| blur-tiled (Fig.3.6) | $S_0 : A_0[ty, tx, x \bmod B, y \bmod B]$<br>$S_1 : A_1[ty, tx, x \bmod B, y \bmod B]$ | $A'_0[ty, tx, (y-2x) \bmod (3B-2)]$<br>$A'_1[ty, tx, x \bmod B, y \bmod B]$ | $\frac{B}{3}$<br>1 | 0.738s |
| unsharp-tiled | $S_0 : A_0[z, ty, tx, x \bmod B, y \bmod B]$<br>$S_1 : A_1[z, ty, tx, x \bmod B, y \bmod B]$ | $A'_0[z, ty, tx, (y-4x) \bmod (5B-4)]$<br>$A'_1[z, ty, tx, -y \bmod B, x \bmod B]$ | $\frac{B}{5}$<br>1 | 1.013s |

Table 5.8: Storage reduction obtained using our approach (SMO) compared to the baseline (successive modulo [LF98] followed by rectangular hull), where $B$ is the loop blocking factor

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 5.7 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.

## 5.3 Storage Mappings Using Enumerative Heuristic

The storage mappings enumerated using Algorithm 2 for various benchmarks are shown in Table. 5.9 and Table 5.10. For each benchmark, we only list up to five different storage mappings that were obtained. Benchmarks with higher dimensionality such as LBMD3Q19 typically had a lot more, although we noticed that some of the later storage mappings enumerated were merely permutations of the earlier ones. It can be seen that searching for alternative storage mappings can be beneficial in some cases. For example, we were thus able to obtain solutions which provide a better storage reduction factor of $\frac{B}{2}$ for the diamond-tile and stencil-1d-hex-tile benchmarks. Furthermore, it is interesting to note that there are alternative storage mappings for the LBM benchmarks which also lead to a storage reduction factor of two. In such cases, where there are multiple storage mappings with same storage reduction factors, it may be beneficial to choose a mapping with relatively simpler access expressions. Due to the exploratory nature of the enumerative heuristic, it is possible to find alternative storage mappings with storage requirements greater than that obtained by choosing the canonical hyperplanes e.g. consider the third storage mapping obtained for stencil-1d-pllgm-tile, shown in Table 5.10, which is worse

than the baseline storage requirement by a factor of two. Such alternative solutions, in fact, justify the selection criteria for storage hyperplanes used in Algorithm 1.

We applied the enumerative approach using Algorithm 4 for the benchmarks with inter-array reuse opportunities as well. However, better storage reductions were not obtained — the solutions found using Algorithm 3 were already exploiting the inter-array reuse opportunities quite well.

| Benchmark | Modulo storage mappings | | Reduction factor |
|---|---|---|---|
| | baseline | SMO | (approx.) |
| produce-consume (Fig.3.1) | $A[t \bmod N, i \bmod N]$ | $A[(i-t) \bmod (2N-1)$ | $\frac{N}{2}$ |
| | | $A[i \bmod N, t \bmod N]$ | 1 |
| | | $A[t \bmod N, i \bmod N]$ | 1 |
| blur-interleaved (Fig.3.3(b)) | $blurx[y \bmod 3, x \bmod N]$ | $blurx[(2x-y) \bmod (2N+1)]$ | $\frac{3}{2}$ |
| | | $blurx[x \bmod N, y \bmod 3]$ | 1 |
| | | $blurx[y \bmod 3, x \bmod N]$ | 1 |
| blur-tiled (Fig.3.6) | $A[ty, tx, x \bmod B, y \bmod B]$ | $A[ty, tx, (y-2x) \bmod (3B-2)]$ | $\frac{B}{3}$ |
| | | $A[ty, tx, y \bmod B, x \bmod B]$ | 1 |
| | | $A[ty, tx, x \bmod B, y \bmod B]$ | 1 |
| harris-corner-tiled | $sobel[tx, ty, x \bmod B, y \bmod B]$ | $sobel[tx, ty, (y-2x) \bmod (3B-2)]$ | $\frac{B}{3}$ |
| | | $sobel[tx, ty, y \bmod B, x \bmod B]$ | 1 |
| | | $sobel[tx, ty, x \bmod B, y \bmod B]$ | 1 |
| unsharp-mask-tiled | $A[z, tx, ty, x \bmod B, y \bmod B]$ | $A[z, tx, ty, (y-4x) \bmod (5B-4)]$ | $\frac{B}{5}$ |
| | | $A[z, tx, ty, -y \bmod B, -x \bmod B]$ | 1 |
| | | $A[z, tx, ty, x \bmod B, -y \bmod B]$ | 1 |
| LBM-D2Q9 (Fig.3.7) | $A[t \bmod 2, i \bmod N, j \bmod N]$ | $A[(i-2t) \bmod (N+2), j \bmod N]$ | 2 |
| | | $A[-t \bmod 2, -i \bmod N, j \bmod N]$ | 1 |
| | | $A[i \bmod N, t \bmod 2, j \bmod N]$ | 1 |
| | | $A[(-t+j) \bmod (N+1), (t-i) \bmod (N+1)]$ | 2 |
| | | $A[(-t+i) \bmod (N+1), (-t+j) \bmod (N+1)]$ | 2 |

Table 5.9: Various modulo storage mappings enumerated using Algorithm 2 compared to the baseline (successive modulo [LF98], where $B$ is the loop blocking factor

| Benchmark | Modulo storage mappings | | Reduction factor |
|---|---|---|---|
| | baseline | SMO | (approx.) |
| LBM-D3Q19 | $A[t \bmod 2, i \bmod N, j \bmod N, k \bmod N]$ | $A[(i-2t) \bmod (N+2), j \bmod N, k \bmod N]$ | 2 |
| | | $A[-t \bmod 2, k \bmod N, j \bmod N, -i \bmod N]$ | 1 |
| | | $A[-i \bmod N, -t \bmod 2, k \bmod N, -j \bmod N]$ | 1 |
| | | $A[(2t-j) \bmod (N+2), (-t+i) \bmod (N+1), -k \bmod N]$ | 2 |
| | | $A[-j \bmod N, (-2t+i) \bmod (N+2), -k \bmod N]$ | 2 |
| diamond-tile (Fig.3.9) | $A_B[tt \bmod B, ii \bmod (2B-1)]$ | $A_B[(tt-3ii) \bmod (6B-5)]$ | $\frac{B}{3}$ |
| | | $A_B[-tt \bmod (B), ii \bmod (2B-1)]$ | 1 |
| | | $A_B[-ii \bmod (2B-1), tt \bmod 2]$ | $\frac{B}{2}$ |
| | | $A_B[(-tt+ii) \bmod (2B-1), ii \bmod B]$ | 1 |
| stencil-1d-pllgm-tile | $A_B[tt \bmod B, ii \bmod B]$ | $A_B[(ii-tt) \bmod (3B-2)$ | $\frac{B}{3}$ |
| | | $A_B[-tt \bmod B, -ii \bmod B]$ | 1 |
| | | $A_B[-ii \bmod (2B-1), -tt \bmod B]$ | $\frac{1}{2}$ |
| | | $A_B[(-tt-ii) \bmod B, tt \bmod B]$ | 1 |
| | | $A_B[(-tt-ii) \bmod B, ii \bmod B]$ | 1 |
| stencil-1d-hex-tile | $A_B[tt \bmod B, ii \bmod (3B-2)]$ | $A_B[(tt-3ii) \bmod (9B-8)]$ | $\frac{B}{3}$ |
| | | $A_B[-tt \bmod B, -ii \bmod (3B-2)]$ | 1 |
| | | $A_B[-ii \bmod (3B-2), -tt \bmod 2]$ | $\frac{B}{2}$ |
| | | $A_B[(tt-ii) \bmod (3B-2), -tt \bmod B]$ | 1 |

Table 5.10: Various modulo storage mappings enumerated using Algorithm 2 compared to the baseline (successive modulo [LF98], where $B$ is the loop blocking factor

# CHAPTER 6

# POLYHEDRAL COMPILATION OF A GRAPHICAL DATAFLOW LANGUAGE

In this chapter, we tackle the problem of applying existing polyhedral techniques for transformation of graphical dataflow programs. Specifically, we deal with the problem of extracting the polyhedral representation of a given graphical dataflow program as well as that of synthesizing the latter given its equivalent polyhedral representation.

## 6.1 Extracting the Polyhedral Representation

The polyhedral representation of a SCoP typically consists of an abstract mathematical description of the iteration domain, schedule and array accesses for each statement. The array accesses are also classified as either read or write accesses. Each of these are expressed as affine functions of the enclosing loop iterators and symbolic constants.

### 6.1.1 Challenges

A graphical dataflow program has no notion of a statement. The program is a collection of nodes that represent specific computations, with data flowing along edges that connect one node to another. Referential transparency ensures that each edge could be associated with its own distinct memory location. Generally, copy-avoidance strategies are used to maximize inplaceness of output and input data. However, the exact memory allocation depends on the specific strategy used. Additionally, the problem of copy-avoidance is closely tied with the problem of scheduling the computation nodes. In the matmul program (Figure 1), consider the array write $u$ and the array read $r$ that share the same data source (say, $v$). If no array copy is to be created, the read must be scheduled ahead of the write, i.e., $u <_s r$ is not *consistent* with $(v, w_1) \rightsquigarrow (w_1, u) \wedge (w_1, u) \rightsquigarrow (u, w_2) \wedge (v, w_1) \rightsquigarrow (w_1, r)$. If the write is scheduled first, the read must work on a copy of the array as the write is likely to overwrite the array input. Abu-Mahmeed et al. [AMMB$^+$09] have looked into the problem of scheduling to maximize the inplaceness of aggregate data. To summarize, the main challenges in the extraction of the polyhedral representation for a graphical dataflow program are as follows:

1. A graphical dataflow program cannot be viewed as a sequence of statements executed one after the other.

2. While the access expressions could be analyzed just like parse trees, it is difficult to relate the access to a particular array definition as the exact memory allocation depends on the specific copy-avoidance strategy used.

3. The actual execution schedule of the computation nodes determined depends on the copy-avoidance decisions.

A trivial polyhedral representation can be extracted by treating each node of a graphical data flow program as a statement analogue while making the conservative assumption that data is copied over each edge. As most compilers make use of copy-avoidance strategies, such a polyhedral representation most certainly over-estimates the amount of data

space required. This also results in an over-estimation of the computation e.g. an array copy. Therefore, the problem of polyhedral transformation in such a representation begins with a serious limitation in terms of dataspace and computation over-estimation. In essence, extraction of a polyhedral representation of a dataflow program part cannot negate the copy-avoidance optimizations. The inplaceness opportunities in the dataflow program must be factored into the analysis.

### 6.1.2 Static Control Dataflow Diagram (SCoD)

A SCoP is defined as a maximal set of consecutive statements without while loops, where loop bounds and conditionals may only depend on invariants within this set of statements. Analogous to this, we now characterize a canonical graphical dataflow program, a Static Control Dataflow Diagram (SCoD), which lends itself well to existing polyhedral techniques for program transformation. The reasoning behind each individual characteristic is provided later.

1. It is a maximal dataflow diagram without constructs for loops that are not countable, where the countable loop bounds and conditionals, in any diagram, only depend on parameters that are invariant for that diagram. Nodes in the SCoD (and its nested diagrams) must be functional, without causing run-time side-effects or relying on any run-time state.

2. The only array primitives that feature as nodes in a SCoD and its nested diagrams are those which read an array element or write to an array element. More importantly, primitives that output array data that cannot be inplace to an input array data cannot be present in the diagrams.

3. For an array data source in any diagram, $(v_1, w)$, there exists at most one node $v_2$ such that $(v_1, w) \rightsquigarrow (w, v_2) \wedge v_2 \times w$ i.e., $v_2$ performs a destructive update.

4. Data flowing into a loop in any diagram is either loop-invariant data or loop-carried scalar data or loop-carried array data that has an associated can-inplace path through

the loop body, which creates the loop-carried dependence, i.e., loop input $x \in I \Rightarrow x \in Inv \vee (x \in ICar \wedge (isScalarType(x, w_x) \vee (isArrayType(x, w_x) \wedge (x, w_x) \rightsquigarrow (w_y, y))))$ where $y = lcd^{-1}(x)$, $w_x$ and $w_y$ the input and output wires.

5. In any diagram, there is no can-inplace path from a loop-invariant data input to the loop-carried data input of an inner loop or to the array input of an array element write node, i.e., in any DAG, $G = (N, E)$ that corresponds to the body of a loop, if $(v_1, w_1)$ is the loop invariant input, then there does not exist any edge $(w_2, v_2)$ such that $(v_1, w_1) \rightsquigarrow (w_2, v_2) \wedge v_2 X w_2$.

The first characteristic is closely tied with the characterization of a SCoP. The rest of the characterization specifies a canonical form of dataflow diagram which has can-inplace relations that facilitate polyhedral extraction. As explained earlier, a naive implementation of a dataflow language could write each new output into a new memory location. The question of whether a particular wire vertex gets a new memory allocation or not depends on the actual copy-avoidance strategy employed by the compiler. The problem of extracting the polyhedral representation of an arbitrary dataflow diagram, therefore depends on the copy-avoidance strategy. In order to make the polyhedral extraction independent of it, we canonicalize the dataflow in a given diagram in accordance with the above characteristics.

An operation such as appending an element to an input array data is a perfectly valid dataflow operation. Clearly, the output array cannot be inplace to the input array. (2) ensures that such array operations are disallowed. Furthermore, it is possible in a dataflow program to overwrite multiple, distinct copies of the same array data. In such a case, a copy-avoidance strategy would inplace only one of the copies with the original data and the rest of them would be separate copies of data. (3) precludes such a scenario. It is important to note that it however, still allows multiple writes. (4) ensures that loop-carried dependence involving array data is tied to a single array data source. Assuming the absence of (5), data flowing from loop-invariant source vertex to a loop-carried input of an inner loop would necessitate a copy because the source data would have a pending read in subsequent iterations of the outer loop.

**Theorem 1.** *In any diagram of the SCoD, G = (V, E), there exists a schedule $<_s$ of the computation nodes V, which is consistent with the conjunction of all possible can-inplace relations, $\bigwedge_{x,\, y,\, z \in V}$ ((x, y) $\rightsquigarrow$ (y, z)), where isArrayType(x, y) $\wedge$ isArrayType(y, z) holds.*

*Proof.* Consider an array data source *(v, w)* in G with $v \in I$. If $v_1, v_2, \ldots, v_n$ are the nodes that consume the array data, then in accordance with characteristic (3), there is at most one node $v_i$ such that *(v, w)* $\rightsquigarrow$ *(w, $v_i$)* $\wedge$ $v_i \times w$. Without any loss of generality, we can assume that $i = n$. This implies that any valid schedule $<_s$ where $v_1 <_s v_2 <_s v_3 \ldots <_s v_n$ holds is consistent with $\bigwedge_{i=1}^{n}$ *((v, w)* $\rightsquigarrow$ *(w, $v_i$))*. Similar scheduling constraints can be inferred for the nodes that consume the array data produced by $v_n$ and so on, thereby ensuring that all the can-inplace relations are satisfied for array dataflow. The new constraints inferred cannot contradict an existing constraint as the graph is acyclic. Therefore, any valid schedule $<_s$ where all the inferred scheduling constraints are satisfied is consistent with maximum array inplaceness in the diagram. □

Essentially, in a SCoD, it is possible to schedule the computation nodes such that no new memory allocation need be performed for any array data inside the SCoD, i.e., all the array data consumed inside the SCoD will then have an inplace source that ultimately lies outside the SCoD.

**Lemma 2.** *In any diagram of the SCoD, G = (V, E), for any sink vertex t $\in$ O, that has array data flowing into it, a can-inplace path exists from a source vertex s $\in$ I to t.*

*Proof.* There must exist a node $v_1$ which produces the array data flowing into $t$ through wire $w$. So, *($v_1$, w)* $\rightsquigarrow$ *(w, t)* holds. In accordance with the model, $v_1$ can either be an array write node or a loop. In either case, there must exist a node $v_2$ which produces the array

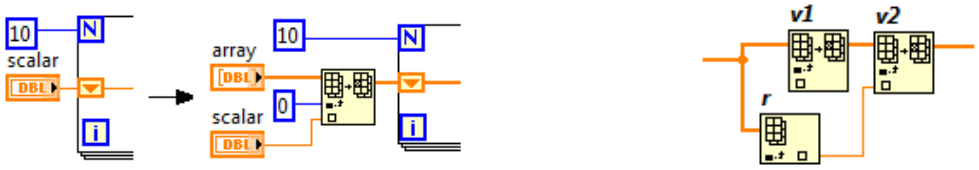
data flowing into $v_1$ and so on until a source vertex $s$ is encountered. The path traversed backwards from $t$ to $s$ clearly constitutes a can-inplace path. □

### 6.1.3 A multi-dimensional schedule of compute-dags

A *compute-dag*, $T = (V_T, E_T)$ in a diagram $G = (V, E)$, is a sub-graph of $G$ where there exists a node, $r \in V_T$ such that for every other $x \in V_T$ there exists a path from $x$ to $r$ in $T$ (the node $r$ will hereafter be referred to as the root node). As it is possible to pick inplace opportunities such that no array data need be copied on any edge in the SCoD, any diagram in the SCoD can be viewed as a sequence of computations that write on the incoming array data. Instead of statements, compute-dags, which are essentially dags of computation nodes can be identified. Consider an array write node or a loop node, both of which can overwrite an input array. Starting with a dag that is just this node as the root, the compute-dag can be built recursively by adding nodes which produce data that flows into any of the nodes in the dag. Such a recursive sweep of the graph stops on encountering another array write or loop node. However, while identifying compute-dags in a diagram, it is necessary to account for all the data produced by the nodes in the diagram.

**Theorem 2.** *In any diagram of the SCoD, $G = (V, E)$, for every edge $(x, y) \in E$ where $x$ is a computation node, there exists a compute-dag $T_i = (V_i, E_i)$ in the set $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ of compute-dags rooted at array write or loop nodes, such that $x \in V_i$ iff only array data flows out of every diagram.*

*Proof.* In accordance with Lemma 1, for any sink vertex $t \in O$, with array data flowing into it, there exists a can-inplace path $p_{s \to t} = \{s, \ldots, v, w, t\}$ from a source vertex $s$ to $t$. Also, every node $x$, which is not dead-code, must have a path $q_{x \to t_j} = \{x, y, \ldots, v_j, w_j, t_j\}$ to at least one sink vertex $t_j \in O$. If only array data flows into every sink vertex, consider




(a) Treating scalar loop-carried data as single-element array

(b) Scheduling compute-dags rooted at $v_1$ and $v_2$ in total order is not possible

Figure 6.1: Single-element arrays and contradiction in schedule of compute-dags.

the first vertex $z$ at which the path $q_{x \to t_j}$ overlaps with $p_{s_j \to t_j}$. $z$ must either be a loop-node or an array write node. In either case $x$ must be part of a compute-dag rooted at $z$ (in the former case, if $x \neq z$, notice also that the data flowing along $(x, y)$ must be the intermediate result of a computation that produces the loop-invariant data for the loop $z$). On the other hand, if scalar data can flow into a sink vertex $t$, clearly, the path from node $x$ to $t$ is not guaranteed to have either an array write node or a loop node. Consequently, the node $x$ is not guaranteed to be part of any compute-dag. Likewise, if loops can have scalar loop-carried data since a scalar loop-carried output is represented as a sink vertex in the loop body DAG. □

In order to address the consequences of Theorem 2, it is necessary to treat scalar data flowing out of the diagram as a single-element array. This results in a compute-dag that accounts for the scalar dataflow. Likewise, loop-carried scalar data must also be treated as single-element array. The dataflow into the loop-carried input is treated as a write to the array resulting in a corresponding compute-dag (refer Figure 6.1(a)). (Hereafter, we assume in the following discussion, that scalar data flowing out of a diagram and loop-carried scalar data are treated specially in this way as single-element arrays).

Each diagram in a SCoD is analyzed for compute-dags, starting from the top-level diagram. Suppose $\theta$ is the scheduling function. At each diagram level, $d$, the set of roots of the compute-dags in $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ are ordered as follows:

- If data produced by a root node $n_1$ is consumed by root node $n_2$, then $\theta^d_{n_1} \prec \theta^d_{n_2}$.

- In accordance with Theorem 1, if there is an array write in a compute-dag rooted at $n_1$ and an array read in a compute-dag rooted at $n_2$, both of which are dependent on the same array data source, then $\theta^d_{n_1} \succ \theta^d_{n_2}$. Scheduling $n_1$ ahead of $n_2$ in the polyhedral representation would be unsafe. Such a schedule would only be possible if $n_2$ were to read a copy of the input array, allowing $n_1$ to overwrite the input array. The safe schedule ensures that an array copy is not required.

- If neither of the above hold for the two root nodes, either $\theta^d_{n_1} \succ \theta^d_{n_2}$ or $\theta^d_{n_1} \prec \theta^d_{n_2}$ should hold true.

Each diagram in the diagram hierarchy of the SCoD contributes to a dimension in the global schedule. Each loop encountered introduces an additional dimension. The total order on the compute-dag roots in any diagram determines the time value at which each compute-dag can be scheduled in that dimension. The global schedule is obtained by appending its time value in the owning diagram to the schedule of the owning loop, if any, together with the loop dimension.

Apart from ensuring that all the data produced by the nodes in a diagram are accounted for, it must also be possible to schedule the compute-dag roots in a total order.

**Theorem 3.** *In any diagram of the SCoD, G = (V, E), it is possible to schedule the set of roots of the compute-dags in $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ in a total order if for every path $p_{v_1 \rightarrow v_2}$ between a pair of roots, $v_1$ and $v_2$, there does not exist an array read node, r in the compute-dag $T(v_2)$, such that r and $v_1$ share the same data source (x, w) with $v_1 \times w$ and a path $q_{r \rightarrow v_2}$ exists that does not include $v_1$ on it.*

*Proof.* Consider a pair of root nodes $v_1$ and $v_2$ (refer Figure 6.1(b)). In accordance with the scheduling constraints specified above, $\theta^d_{v_1} \prec \theta^d_{v_2}$ if a path $p_{v_1 \rightarrow v_2}$ exists in G. This scheduling order is contradicted only if for some reason $\theta^d_{v_1} \succ \theta^d_{v_2}$ must hold, which can happen only if the compute-dag $T(v_2)$ contains a node that must be scheduled ahead of $v_1$,

i.e., an array read node that shares the same source as $v_1$. If such a node does not exist, then the contradiction never arises leading to a total ordering of the compute-dag roots. Similarly, there is no contradiction in schedule order if a path $p_{v_1 \to v_2}$ does not exist. □

In order to address the consequence of Theorem 3, while building compute-dag rooted at $v_2$, it is also necessary to stop on encountering an array read node $r$ when there is a node $v_1$ with the same array source, overwriting the incoming array, such that there exists a path from $r$ to $v_2$ which does not include $v_1$. A separate compute-dag rooted at such an array read must be identified, thereby breaking the compute-dag that would have been identified otherwise (rooted at $v_2$) into two different dags.

The set of actual statement analogues is $\Sigma = \{T_1, T_2, \ldots, T_n\}$ such that the *root*($T_i$) for any $T_i \in \Sigma'$ is not a loop. Algorithm 5 provides a procedure for identifying the set of statement analogues in a given dataflow graph G = (V, E) of a particular diagram. It is possible for two statement analogues to have common sub-expressions. However, the nodes in a SCoD are functional, making the common sub-expressions also so.

**Analysis of iteration domains.** We assume that loop normalization has been done, i.e., all for-loops have a unit stride and a lower bound of zero. Analyzing the iteration domain of a for loop only involves the analysis of the dataflow computation tree that computes the upper bound of the for loop. This analysis is very similar to parsing an expression tree. Symbolic constants are identified as scalar data sources that lie outside the SCoD. Loop iterators and constant data sources are explicitly represented as nodes in our model.

**Analysis of array accesses.** The access expression trees for the array reads and array writes which are present in the compute-dags of the statement analogues are analyzed to obtain the access functions. The most important problem of tying the array access to a particular memory allocation is resolved easily. Due to a carefully determined scheduling order, which schedules array reads ahead of an array write having the same source, all the accesses can be uniquely associated with array data sources that lie outside the SCoD. This is regardless of the actual copy-avoidance strategy that may be used. Additionally, the scalar data produced by an array read that is the root of its own compute-dag and

**Algorithm 5** *identify-compute-dags(G = (V, E))*

**Require:** Treat scalar data flowing out of diagram as single element arrays

**Require:** Loop-invariant computations have not been code-motioned out into an enclosing diagram

```
1:  procedure IDENTIFY-COMPUTE-DAGS(G = (V, E))
2:      Σ = ∅
3:      for all n ∈ V | isArrayWriter(n) ∧ !isloop(n) do
4:          Σ = Σ ∪ build-compute-dag(n, G)        ▷ compute-dag from G, with root n
5:      for all n ∈ V | root-candidate[n] do
6:          Σ = Σ ∪ build-compute-dag(n, G)
7:      return Σ

8:  procedure BUILD-COMPUTE-DAG(n, G = (V, E))
9:      V_T = {n}, E_T = ∅
10:     while (x, y) = get-new-node-for-dag(n, T = (V_T, E_T), G) do
11:         V_T = V_T ∪ x, E_T = E_T ∪ {(x, y)}
12:     return (V_T, E_T)

13: procedure GET-NEW-NODE-FOR-DAG(n, T = (V_T, E_T), G = (V, E))
14:     for each (x, y) ∈ E do
15:         if (x, y) ∉ E_T ∧ y ∈ V_T ∧ !isArrayWriter(x) ∧ !isloop(x)
16:             if isArrayReader(x)
17:                 z = get-array-write-off-same-source-if-any(x)
18:                 if there exists a path p_{z→n}
19:                     root-candidate[x] = true
20:                     continue
21:             return (x, y)
22:     return ∅
```

a node in another compute-dag is treated as a single-element array, thereby encoding the corresponding dependence in the array accesses of both the compute-dags. So, each

statement analogue has exactly one write access.

## 6.2 Code Synthesis

A polyhedral optimizer can be used to perform the required program transformations on the polyhedral representation of the SCoD. We now consider the problem of synthesizing a SCoD given its equivalent polyhedral representation.

### 6.2.1 Input

The input polyhedral representation must capture the iteration domain, access and scheduling information of the statement analogues i.e., the set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags, which are also available as input. Each compute-dag, derived perhaps from an earlier polyhedral extraction phase, has exactly one array write node, which is the root of the dag.

The polyhedral representation must have identity schedules. Any polyhedral representation with non-identity schedules can be converted to one with identity schedules by performing code generation and extracting the generated code again into the polyhedral representation. In this manner, scheduling information gets into statement domains and the schedule extracted from the generated code is an identity one. Once an equivalent polyhedral representation in this form has been obtained, the approach described in the rest of this section is used to synthesize a SCoD.

### 6.2.2 Synthesizing a Dataflow Diagram

The pseudo-code for synthesizing a dataflow diagram is presented in Algorithms 7 and 6. The statement analogues are processed in their global schedule order (line 6.5). The iteration domain and scheduling information of a statement analogue are together used to create the surrounding loop-nest (line 6.7). Lower and upper bounds are inferred for each loop iterator. In case the for-loop is a normalized for-loop as in our abstract model,

**Algorithm 6** *Synthesize-SCoD()*

1: **Convention:** If $s$ represents a source vertex, the paired sink is $s'$
2: **procedure** SYNTHESIZE-SCOD( )
3: Let $G_0$ be the DAG of the top level diagram, $G_0 = (\emptyset, \emptyset)$
4: *create-source-vertex-for-each-global-parameter($G_0$)*
5: **for** each statement analogue, $T$ in global schedule order **do**
6: Read *domain (D)*, *identity schedule ($\theta$)* and *access (A)* matrices
7: $l$ = *create-or-get-loop-nest($G_0$, D, $\theta$)* ▷ $l$, innermost loop
8: *add-compute-dag(l, T, $G_0$)*
9: **for** each (variable, read access) pair *(v, a)* in $A$ **do**
10: $(s_0, s_0')$ = *create-source-and-sink-vertices-if-none(v, $G_0$)*
11: *create-or-get-dataflow($s_0$, $s_0'$, l, $G_0$, READ)*
12: *array-read-node-access(a, T, l, $G_0$)* ▷ node reads data flowing into $l$ through loop-invariant input or data flowing into the loop carried output if it exists. Create array index expression tree using $a$
13: *(v, a)* = *get-variable-write-access-pair(A)*
14: $(s_0, s_0')$ = *create-source-and-sink-vertices-if-none(v, $G_0$)*
15: *create-or-get-dataflow($s_0$, $s_0'$, l, $G_0$, WRITE)*
16: *insert-array-write-node(a, T, l, $G_0$)* ▷ node is added to flow path so that it overwrites the data flowing into the loop-carried output of $l$. Create array index expression tree
17: *create-dataflow-from-parameters-and-iterators(c, $G_0$)*
18: **return** $G_0$

the actual upper bound will be a difference of the minimum and maximum of the inferred upper and lower bounds plus one. Built-in primitives for various operations such as max, min, floor, ceil etc. may be used to set up the loop-control. Note that if the required loop-nest has been created already for a statement analogue scheduled earlier, it need not be created again. The compute-dag is then added to the dataflow graph of the enclosing loop (line 6.8).

**Inherent parallelism – the factor to consider.** Dataflow programs are inherently parallel. A computation node is ready to be fired for execution as soon as all its inputs are

**Algorithm 7** *Creation of loop-carried and loop-invariant dataflow*

**procedure** CREATE-OR-GET-DATAFLOW($s_0, s'_0, l_m, G_0, access-type$)
  $\{l_1,...,l_m\} = get\text{-}enclosing\text{-}loops(l_m)$ ▷ $\{G_1,...,G_m\}$ be their DAGs
  $sources = get\text{-}inflow\text{-}if\text{-}any(s_0, l_m, G_0)$
  $c = \max i \mid i \in \{0, 1,...,|sources|\}$ and $s_i \in ICar[l_i]$ for $i > 0$
  **if** *access-type* $==$ *WRITE*
    **if** $c < m$
      find $v \mid (v, w), (w, s'_c) \subset E_{|c|}$
      create flow path from $v$ to $s'_c$ through loop $l_m$ via loop-carried inputs/outputs (transforming $s_{c+1},...,s_{|sources|}$ into loop-carried inputs)
      replace flow path $(v, w, s'_c)$ with this new flow path
  **else if** $|sources| < m$ ▷ must be a read access
    **if** $c == |sources|$ ▷ use data overwritten in outer loop
      find $v \mid (v, w), (w, s'_c) \subset E_{|c|}$
    **else** $v = s_{|sources|}$ ▷ extend loop-invariant flow
    create a flow path through loop-invariant inputs from $v$ to $l_m$
  **return**

**procedure** GET-INFLOW-IF-ANY($s_0, l_m, G_0$)
  $\{l_1,...,l_m\} = get\text{-}enclosing\text{-}loops(l_m)$ ▷ $l_1$ outermost
  $s = s_0$, $H = G$, $U = V$, $F = E$, $sources = \emptyset$
  **for** $i \leftarrow 1, m$ **do**
    $w_i =$ wire carrying data from $s$
    **if** $\exists\, w \in U \mid (w, l_i) \in F \wedge (s, w_i) \rightsquigarrow (w, l_i)$
      $H =$ DAG that describes body of loop $l_i$
      $s =$ source vertex in $H$ that corresponds to loop input $(w, l_i)$
      $sources =$ append $s$ to the *sources* list
    **else break**
  **return** *sources*

available. It is essential to exploit this inherent parallelism during code synthesis. In order to infer such parallelism and exploit it, we reason in terms of *coalesced dependences*. A coalesced dependence is the same as a regular data dependence except that two accesses are considered to be in conflict if they even access the same variable (potentially an aggregate data type), as opposed to the same location in the aggregate data. For example, an array access that writes to odd locations does not conflict with another that reads from even locations. However, a coalesced dependence exists between the two. Analogous to regular data dependences, we now also use the terms flow, anti, and output coalesced dependences.

A unique source-sink vertex pair $(s_0, s_0')$ is created in the top-level DAG, $G_0$, of the top-level diagram for each variable $v$ whose access is described in the access matrices (lines 6.10, 6.14). A dataflow path is also created from $s_0$ to $s_0'$. The problem of synthesizing a dataflow diagram is essentially a problem of synthesizing the dependences between the given set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags in terms of edges that will connect them together. Specifically, as all the dependences involve array variables (may be single-element), these interconnecting edges represent the dataflow between array read or write nodes in the compute-dags, through intervening loops. Consider set of array write nodes $U = \{u_1, u_2, \ldots, u_n\}$, which correspond to write accesses on the same variables in a particular time dimension such that $u_i$ is scheduled ahead of $u_j$ for all $i < j$ (i.e., the corresponding compute-dags).

**Theorem 4.** *All coalesced output dependences on a variable in the polyhedral representation are satisfied by a synthesized dataflow diagram if in any diagram, all array write nodes $u_1, u_2, \ldots, u_n$ corresponding to write accesses to that variable lie on the same can-inplace path $p_{u_1 \to u_n}$.*

*Proof.* Suppose all the nodes in $U = \{u_1, u_2, \ldots, u_n\}$ are scheduled in the outermost diagram. A coalesced output dependence exists between any pair of write nodes scheduled in

this diagram, thereby defining a total ordering on the set $U$. Therefore, all the corresponding array write nodes must be inserted along the can-inplace path $p_{s_0 \to s_0'}$. Now consider a write node $u$ scheduled in an inner loop. A coalesced output dependence exists between $u$ and any array write node $u_i \in U$. This is ensured by inserting the inner loop along the path $p_{s_0 \to s_0'}$, in accordance with its schedule order relative to the other write nodes on the path. The incoming and outgoing edges of the loop node on the can-inplace path must correspond to the loop-carried input and its paired output, which in turn serve as the source and sink vertices in the DAG of the loop body. □

**Theorem 5.** *A coalesced flow dependence in the polyhedral representation is satisfied by a synthesized dataflow diagram if the array write node and read node associated with the dependence lie on the same can-inplace path.*

*Proof.* Each of the array write nodes $u_1, u_2, \ldots, u_m$ lies on the can-inplace path $\mathrm{p}_{u_1 \to u_m}$ due to Theorem 4. A coalesced flow dependence exists between the write access $u_m$ and read access $r$. Therefore, there must be a path $\mathrm{p}_{u_m \to r}$, which means that all of these nodes must lie on the same can-inplace path $\mathrm{p}_{u_1 \to r}$. If a read access $r$ is the only access to a variable inside an inner loop $l$, the coalesced flow dependence between $r$ and any $u_i$ scheduled earlier is satisfied by a can-inplace path $p_{u_1 \to l}$. The incoming edge to $l$ on this path need only correspond to a loop-invariant input. It acts as a data source for $r$ in the loop body. □

Together, from Theorem 4 and Theorem 5, it can be seen that the path $p_{u_1 \to r}$ diverges from the path $\mathrm{p}_{u_1 \to u_n}$ at $u_m$ i.e., the last write scheduled ahead of $r$. This enables the concurrent execution of the array write node $u_{m+1}$ and $r$, thereby exposing the inherent parallelism in a dataflow diagram discussed earlier. There is no coalesced output or coalesced flow dependence between $u_{m+1}$ and $r$. Also, just as the output array of an array write node can be inplace to the input array, loop-carried array outputs of a loop node can be inplace to the corresponding input. Similarly, a loop-invariant array input corresponds

to the array input of a read node, as they do not have a corresponding output that can be inplace. Due to this symmetric relationship, based on coalesced dependences, we can infer inherent parallelism in the following scenarios:

- Consider two compute-dags, $T_1$ and $T_2$, scheduled in the same time dimension, $d$, such that $\theta^d_{T_1} \prec \theta^d_{T_2}$ with no coalesced output or coalesced flow dependence between them e.g. the two compute-dags have array accesses on disjoint sets of arrays. $T_1$ and $T_2$ then constitute two tasks that can be executed in parallel in a dataflow program.
- Consider two loops, $l_x$ and $l_y$, scheduled in the same time dimension such that there is no coalesced output or coalesced flow dependence between compute-dags in one loop and those of the other e.g. compute-dags in $l_x$ only read a particular array variable, whereas those in $l_y$ only write to it. The two loops can be executed as parallel tasks. This can be particularly crucial in obtaining good performance.
- Similarly, a loop and a compute-dag scheduled in the same time dimension with no coalesced output or coalesced flow dependence between the compute-dag and those in the loop.

Note that coalesced anti-dependences do not inhibit parallelism. The read and write access on the same variable may share the same data source. The read access can be performed on a copy of the data, while the write access is performed on the source data.

A dataflow diagram synthesized as described in the proofs for Theorem 4 and 5 is indeed a SCoD. The characteristics (1) and (2) are trivially satisfied. The dataflow diagram also meets characteristic (3) as all the array write nodes are serialized in accordance with Theorem 4. Furthermore, the construction described in the proof for Theorem 4 also ensures that whenever a loop-carried input-output pair is created, the corresponding source and sink vertices have an associated can-inplace path, thereby ensuring characteristic (4). Finally, the proof for Theorem 5 also implies a loop-carried input for a particular variable access is created on a loop only when all the accesses to a variable inside the loop-nest are read accesses. Therefore, a flow path from a loop-invariant source vertex to a loop-carried input never exists, ensuring characteristic (6).

Algorithm 6 processes the read accesses of a statement analogue first and then the write access. Algorithm 7, briefly explained below, describes the creation of the array dataflow paths for the corresponding read and write nodes in the compute-dag.

**Read accesses:** Suppose the array read node is scheduled to execute in loop $l_m$. The closest enclosing loop $l_c$ that has an array write node (for a write access on the same variable) in its body, and therefore, an associated loop-carried input $s_c$ is found (line 7.4). A dataflow through loop-invariant inputs is then created to propagate the data flowing into the loop-carried output $s'_c$ (line 7.12) to the inner loop $l_m$ (line 7.14). This is the data produced by the write node associated with the last write access on the variable. However, if part of such a flow through loop-invariant inputs already exists for an intervening loop, it is extended to reach $l_m$ (line 7.13).

**Write accesses:** Suppose the array write node is scheduled in a loop $l_m$. As in the case of a read access, the loop $l_c$ is found (line 7.4). Any flow of data through loop-invariant inputs of intervening loops, from the source $v$ of the loop-carried output $s'_c$ is transformed to a flow of data through loop-carried inputs to the inner loop $l_m$. The newly created data flow through loop-carried inputs and outputs replaces the existing flow path $(v, w, s'_c)$ (line 7.7-line 7.9).

Once the dataflow from the variable source vertex is created to the loop enclosing the access node, it can read or write the data flowing in. The required access computation trees are created using the access information (usually represented by a matrix). The data outputs from these trees serve as the index inputs to access node.

**Loop iterators and global parameters:** Besides the variable accesses, considered so far, there might still be other nodes whose input dataflow is yet to be created. The sources of these node inputs are either the loop iterators or global parameters for the SCoD e.g. consider the compute-dag that corresponds to `(b[i] = a[i] + i)`, the `i` input to the add node in the compute-dag still needs an input dataflow. Two mappings, `paramSource` and `iteratorSource`, from the set of node inputs to the sets of global parameters and loop iterators, can be used to create the input dataflow from the corresponding source vertices. In an actual implementation, these mappings have to be derived from the earlier phases

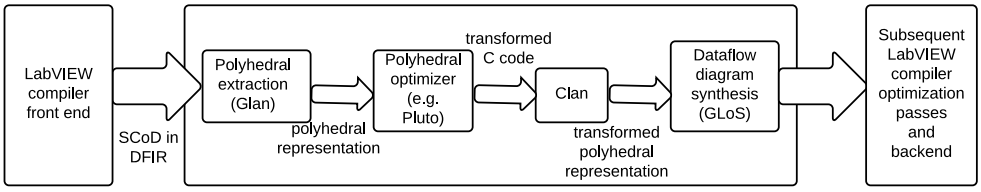



Figure 6.2: A high-level overview of PolyGLoT

of polyhedral extraction and optimization.

## 6.3 The PolyGLoT Auto-Transformation Framework

We employed the techniques described so far to build PolyGLoT, a polyhedral automatic transformation framework for LabVIEW. The LabVIEW compiler translates the source dataflow diagram into a hierarchical, graph-based dataflow intermediate representation (DFIR). Several standard compiler optimizations are performed on this intermediate representation. We implemented a separate pass that uses PolyGLoT to perform polyhedral extraction, auto-transformation and dataflow diagram synthesis in that order. The optimized DFIR graph is then translated to the LLVM IR, which is then further optimized by the LLVM backend compiler that finally emits the machine code.

PolyGLoT consists of four stages. The first stage extracts the polyhedral representation from a user-specified SCoD using the techniques described in Section 6.1. The translation is performed on DFIR. Glan (named after its C counterpart), a G loop analysis tool, was implemented to serve this purpose. The polyhedral representation extracted is used as an input to Pluto, an automatic parallelizer and locality optimizer. Pluto then applies a sequence of program transformations that include loop interchange, skewing, tiling, fusion, and distribution. Pluto internally calls into CLooG to output the transformed program as C code. Glan was used to produce a representative text (encoding a compute-dag id and also text describing the array accesses) for each compute-dag. Thus, we ensured that the transformed C code produced by Pluto included statements that could be matched with the computed-dags identified during extraction.

Clan was used to extract the polyhedral representation of the transformed C code,

which was finally used as the input for GLoS (G loop synthesis). GLoS is a tool that synthesizes DFIR from the input polyhedral representation as per techniques developed in Section 6.2. Pluto was also used to produce scheduling information of loops that it would parallelize using OpenMP. This information was used by GLoS to parallelize the corresponding loops in the synthesized DFIR using the LabVIEW parallel for loop feature.

## 6.4 Experimental Evaluation

For the purpose of experimental evaluation, we implemented many of the benchmarks in the publicly available Polybench/C 3.2 [PBE] suite in LabVIEW. The matmul and ssymm benchmarks from the example test suite in Pluto were also used. Each of these benchmarks were then compiled using five different configurations.

- *lv-noparallel* is the configuration that simply uses the LabVIEW production compiler. This is the baseline for configurations that do not parallelize loops.
- *pg-loc* uses the LabVIEW compiler but with our transformation pass enabled to perform locality optimizations.
- *lv-parallel* again uses the LabVIEW production compiler, but with loop parallelization. The parallel for loop feature in LabVIEW [YFDB10, BDYF10] is used to parallelize loops when possible in the G code.
- *pg-par* is with our transformation pass enabled to perform auto-parallelization but without any locality optimizing transformations. In order to realize a parallel loop identified as parallelizable, Bordelon et al. [BDYF10]'s solution is used. The parallel loops in the transformed code are identified using Pluto [Plu].
- *pg-loc-par* is with our transformation pass enabled to perform both locality optimizations and auto-parallelization.

The comparisons of the runtime performance with various configurations can be found in Table 6.1. The performance numbers were obtained on a dual-socket Intel Xeon CPU

| Benchmark | Problem size | Execution time (seq) | | Speedup | Execution time (8 cores) | | | Speedup over *lv-par* | |
|---|---|---|---|---|---|---|---|---|---|
| | | *lv-nopar* | *pg-loc* | (local) | *lv-par* | *pg-par* | *pg-loc-par* | *pg-par* | *pg-loc-par* |
| atax | NX=4096, NY=4096 | 0.456s | 0.567s | 0.80 | 0.707s | 0.642s | 0.167s | 1.10 | 4.23 |
| bicg | NX=4096, NY=4096 | 0.409s | 0.689s | 0.59 | 0.409s | 0.220s | 0.093s | 1.86 | 4.40 |
| doitgen | NQ=NR=NP=128 | 7.476s | 7.344s | 1.02 | 0.976s | 0.999s | 0.934s | 0.98 | 1.04 |
| floyd-warshall | N=1024 | 86.06s | 91.89s | 0.94 | 82.76s | 13.64s | 4.909s | 6.07 | 16.9 |
| gemm | NI=NJ=NK=1024 | 60.40s | 24.20s | 2.50 | 7.026s | 5.473s | 3.628s | 1.28 | 1.94 |
| gesummv | N=4096 | 0.488s | 0.536s | 0.91 | 0.078s | 0.069s | 0.074s | 1.13 | 1.05 |
| matmul | N=2048 | 688.5s | 196.3s | 3.51 | 89.49s | 94.70s | 27.44s | 0.94 | 3.26 |
| mvt | N=4096 | 1.248s | 0.828s | 1.51 | 0.195s | 0.334s | 0.105s | 0.58 | 1.86 |
| seidel | N=1024, T= 1024 | 44.82s | 44.79s | 1.00 | 45.03s | 9.797s | 8.364s | 4.60 | 5.38 |
| ssymm | N=2048 | 122.8s | 177.4s | 0.69 | 15.03s | 55.45s | 23.85s | 0.27 | 0.63 |
| syr2k | NI=1024, NJ=1024 | 34.03s | 30.86s | 1.10 | 4.190s | 4.423s | 4.223s | 0.95 | 0.99 |
| syrk | NI=1024, NJ=1024 | 24.44s | 22.01s | 1.11 | 2.974s | 3.118s | 2.793s | 0.95 | 1.06 |
| trmm | N=2048 | 231.7s | 64.62s | 3.59 | 41.29s | 39.94s | 11.42s | 1.03 | 3.62 |

Table 6.1: Summary of performance (sequential and parallel execution on an 8-core machine)

E5606 (2.13 GHz, 8 MB L3 cache) machine with 8 cores in all, and 24 GB of RAM. LLVM 2.8 was the final backend compiler used by LabVIEW.

Table 6.1 shows that the benchmarks gemm, matmul, mvt, syr2k, syrk and trmm benefit from locality-enhancing optimizations, in particular, loop tiling [Xue00], and in addition, loop fusion and other unimodular transformations [ASUL06, Wol95]. Table 6.1 also shows the effect of locality optimizations in conjunction with loop parallelization. It can be seen that for floyd-warshall and seidel, loop skewing exposes loop parallelism that could not have been exploited without it. The benchmarks, atax, bicg, floyd-warshall, gemm, matmul, mvt, seidel, syrk, trmm benefit from more coarse-grained parallelism, i.e., a reduced frequency of shared-memory synchronization between cores as a result of loop tiling. In some cases, we see a slow down with PolyGLoT, often by about 10%. We believe that this is primarily due to transformed code generated by PolyGLoT not being optimized by subsequent passes within LabVIEW and the backend compiler (LLVM) as well as the baseline (*lv-noparallel* and *lv-parallel*). This is also partly supported by the fact that *pg-loc* itself produces this slow down, for example, for ssymm. Better downstream optimization within LabVIEW and in LLVM after PolyGLoT has been run can address this. In addition, the loop fusion heuristic used by Pluto can be tailored for LabVIEW code to obtain better performance. Overall, we see a mean speedup of 2.30× with PolyGLoT (*pg-loc-par*) over the state-of-the-art (*lv-parallel*).

## 6.5 Related Work

A significant amount of work has been done on using polyhedral techniques in the compilation of imperative languages [Fea92a, Fea92b, Gri04, BBK+08]. Clan is a widely used research tool for extracting a polyhedral representation from C static control parts [Bas]. A more recent work is `pet` that uses a full-fledged C frontend (Clang for LLVM) [VG12]. Production compilers with polyhedral framework implementations include IBM XL [BGDR10], RSTREAM [MVW+11], and LLVM [GZA+11].

Ellmenreich et al. [ELG99] have considered the problem of adapting the polyhedral

model to parallelize a functional program in Haskell. The source program is analyzed to obtain a set of parallelizable array definitions. Dependence analysis on each array set is then performed to parallelize all the computations within the set.

Johnston et al. [JHM04] review the advances in dataflow programming over the decades. Ample work has been done on parallelizing dataflow programs. It includes the work on loop parallelization analysis by Yi et al. [YFDB10]. Dependences between array accesses are analyzed using standard techniques to determine if a given user-specified loop in a graphical dataflow program can be parallelized. In contrast to these works, the focus of our work is not really on parallelization but on leveraging existing polyhedral compilation techniques to perform dataflow program transformations. Parallelism detection is but a small component of a loop-nest optimization framework. The complete polyhedral representation that we extract from a given dataflow program part can be used to drive automatic transformations, many of which can actually aid parallelization. Furthermore, to the best of our knowledge, no prior art exists that tackles this problem and the problem of dataflow program part synthesis from an equivalent polyhedral representation by exploiting the inplaceness opportunities that can be inferred from the dataflow program. The work of Yi et al. [YFDB10] is commercially available as the parallel for loop feature in LabVIEW, and we compared with it through experiments in Section 6.4. Given an iterative construct in a dataflow program that is marked parallel, Bordelon et al. [BDYF10] studied the problem of parallelizing and scheduling it on multiple processing elements. Our system uses it to eventually realize parallel code from the transformed DFIR.

The interplay between scheduling and maximizing the inplaceness of aggregate data has been studied by Abu-Mahmeed et al. [AMMB$^+$09]. Recently, Gerard et al. [GGPP12] have built on this work to provide a solution for inter-procedural inplaceness using language annotations that express inplace modifications. The soundness of such an annotation scheme is guaranteed by a semi-linear type system, where a value of a semi-linear type can be read multiple times and then updated once. For any array data source in any diagram of the SCoD, there is at most one node that can overwrite it. During the polyhedral extraction, by scheduling a write node after all the read nodes which share the same data

source, we in effect choose semi-linear type semantics on the array data in the dataflow diagram. It also allows us to infer an inplace path of array updates. The inplace path is used for associating the accesses to an array definition in the polyhedral representation, which can have multiple write accesses to the same definition.

# CHAPTER 7

# CONCLUSIONS

Automatic solutions to the storage optimization problem, be it intra-array or inter-array, 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. We cast the problem as one of array space partitioning where each partition uses the same memory location. This allowed us to develop algorithms to find the right orientations for the array partitioning hyperplanes. The algorithms can handle non-convex conflict relations described as union of polyhedra. The algorithm for intra-array reuse is driven by the two objectives of maximizing conflict satisfaction and minimizing conflict distances. For numerous examples and real-world problems, we showed significant reductions in storage requirement over previous techniques, ranging from a constant factor to asymptotic in loop blocking factor or array extents — the latter being a dramatic improvement for practical purposes. While the greedy nature of conflict satisfaction does not necessarily lead to optimal solutions, experimental evaluation shows that it is capable of finding optimal affine mappings in practice. Furthermore, 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 have 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.

Polyhedral optimizers can facilitate full-fledged automatic program transformations by cobbling together machinery drawn from a wide range of polyhedral tools – loop analyzers such as Clan [Bas], Pet [VG12], loop optimizers such as Pluto [Plu], various polyhedral analysis and code generation tools such as ISL [Ver10], Piplib [PIP, Fea88] and CLooG [Bas04]. However, no polyhedral storage optimizer is yet to feature in any standard polyhedral optimization framework. We believe that SMO [SMO16], which is open source and publicly available for download, can serve as a good starting point towards addressing this missing link. Our experience in developing and evaluating SMO has shown that there are also significant challenges in the auto-generation of the transformed array accesses. It would be interesting to investigate the interplay between auto-vectorization opportunities and modulo accesses. Furthermore, we need to explore ways for optimizing the numerous conditional expressions that result from total expansion as well as the modulo operations.

We have tackled the problem of extracting polyhedral representations from graphical dataflow programs that can be used to perform high-level program transformations automatically. Additionally, we also studied the problem of synthesizing dataflow diagrams from their equivalent polyhedral representation. To the best of our knowledge, this is the first work which deals with these problems, and does this while exploiting inplaceness opportunities inherent in a dataflow program. We also demonstrated that our techniques are of practical relevance by building an automatic transformation framework for the LabVIEW compiler that uses them. In several cases, programs compiled through our framework outperformed those compiled otherwise by significant margins. In future, we would like to expand the class of dataflow programs which can be compiled using our framework.

# BIBLIOGRAPHY


[ABD07] Christophe Alias, Fabrice Baray, and Alain 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.

[Ali07] Christophe Alias. Bee+Cl@k, 2007. Bee+Cl@k tool: http://compsys-tools.ens-lyon.fr/bee/.

[AMMB+09] Samah Abu-Mahmeed, Cheryl McCosh, Zoran Budimli, Ken Kennedy, Kaushik Ravindran, Kevin Hogan, Paul Austin, Steve Rogers, and Jacob Kornerup. Scheduling tasks to maximize usage of aggregate variables in place. In *Intl. Conference on Compiler Construction (CC)*, pages 204–219, 2009.

[ASUL06] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools Second Edition*. Prentice Hall, 2006.

[Bas] Cédric Bastoul. Clan: The Chunky Loop Analyzer. The Clan User guide.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, September 2004.

[BB13] Somashekaracharya G. Bhaskaracharya and Uday Bondhugula. PolyGLoT: A Polyhedral Loop Transformation Framework for a Graphical Dataflow Language. In *Intl. conference of Compiler Construction (CC, part of ETAPS)*, pages 138–146, Rome, Italy, March 2013.

[BBC16a] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic Storage Optimization for Arrays. *ACM Transactions on Programming Languages and Systems*, 38(3):11:1–11:23, April 2016.

[BBC16b] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 526–538, 2016.

[BBK+08] Uday Bondhugula, M. Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*, pages 132–146, Apr 2008.

[BDYF10] Adam Bordelon, Robert Dye, Haoran Yi, and Mary Fletcher. Automatically creating parallel iterative program code in a data flow program. (20100306733), December 2010. http://www.freepatentsonline.com/y2010/0306733.html.

[BGDR10] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 343–352, 2010.

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *SC*, pages 40:1–40:11, 2012.

[CC04] Patrick Carribault and Albert Cohen. Application of storage mapping optimization to register promotion. In *ACM International Conference on Supercomputing (ICS'04)*, pages 247–256, St-Malo, France, June 2004.

[CDRV97] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithm. *Parallel Computing*, 23(1-2):251–266, 1997.

[CFGV09] Philippe Clauss, Federico Javier Fernandez, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. VLSI Syst.*, 17(8):983–996, 2009.

[DIY16a] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. Extended lattice-based memory allocation. *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 218–228, 2016.

[DIY16b] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. Liveness analysis in explicitly-parallel programs. *Proceedings of the 6th International Workshop on Polyhedral Compilation Techniques, IMPACT*, Jan 2016.

[DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[ELG99] Nils Ellmenreich, Christian Lengauer, and Martin Griebl. Application of the polytope model to functional programs. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 219–235, 1999.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part II, multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[GCH+14] Tobias Grosser, Albert Cohen, Justin Holewinski, P Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization*, page 66. ACM, 2014.

[GCM97a] Eddy De Greef, Francky Catthoor, and Hugo De Man. Array placement for storage size reduction in embedded multimedia systems. *International Conference on Application Specific Systems, Architectures, and Processors*, pages 66–75, 1997.

[GCM97b] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23(12):1811–1837, 1997.

[GGPP12] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 51–60, 2012.

[GNU10] GNU. GNU Linear Programming Kit (GLPK), 2010.

https://www.gnu.org/software/glpk/.

[Gri04] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[GV15] Pieter Ghysels and Wim Vanroose. Modeling the performance of geometric multigrid stencils on multicore computer architectures. *SIAM J. Scientific Computing*, 37(2):C194–C216, 2015.

[GZA+11] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simburger, Armin Größlinger, and Louis-Noël Pouchet. Polly: Polyhedral optimization in LLVM. In *IMPACT*, 2011.

[HS88] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proceedings of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[Int13] Intel. Using Intel VTune Amplifier XE To Tune Software on the Intel Xeon Processor E5 family, 2013. https://software.intel.com/en-us/articles/using-intel-vtune-amplifier-xe-to-tune-software-on-the-intel-xeon-processor-e5-family.

[Int15] Intel. Intel VTune Amplifier XE 2015 (build 367957), 2015. https://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-release-notes.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1), March 2004.

[Lab10] NI LabVIEW Compiler: Under the Hood, 2010.

http://www.ni.com/white-paper/11472/en.

[LF98] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[LLL01] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 103–112, 2001.

[MCT96] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday 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.

[MVW+11] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-stream compiler. In *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[PAVB15] Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations. *ACM Transactions on Architecture and Code Optimization (TACO)*, May 2015.

[PBE] Polybench. http://polybench.sourceforge.net.

[Pik02] G. Pike. *Reordering and Storage Optimizations for Scientific Programs*. PhD thesis, University of California Berkeley, 2002.

[PIP] PIP: The Parametric Integer Programming Library. http://www.piplib.org.

[Plu] PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[QR00] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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.

[SCFS98] M. Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Intl. conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 24–33, 1998.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[SMO16] A Storage Optimization Tool For Regular Loop Nests, 2016. https://github.com/bondhugula/smo.

[Suc01] S. Succi. *The Lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

[TVA07] William Thies, Frédéric Vivien, and Saman Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Trans. Program. Lang. Syst.*, 29(6), October 2007.

[TVSA01] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman P. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM SIGPLAN symposium on Programming Languages Design and Implementation*, pages 232–242, 2001.

[Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327, pages 299–302. Springer, 2010.

[VG12] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Impact 2012, 2nd International Workshop on Polyhedral Compilation Techniques*, January 2012.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[WR96] Doran Wilde and Sanjay V. Rajopadhye. Memory reuse analysis in the polyhedral model. In *International Euro-Par Conference on Parallel Processing*, pages 389–397, 1996.

[Xue00] Jingling Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[YFDB10] Haoran Yi, Mary Fletcher, Robert Dye, and Adam Bordelon. Loop parallelization analyzer for data flow programs. (20100306753), December 2010. http://www.freepatentsonline.com/y2010/0306753.html.