Uniformization of Affine Dependence Programs for Parallel Embedded System Design

M. Manjunathaiah
OUCL, Park Road
Oxford, OX1 3QD, UK
manju@comlab.ox.ac.uk

G. M. Megson
Dept. of Computer Science,
U. of Reading Whiteknights
PO Box 225, Reading
Berkshire, RG6 6AH, UK
G.M.Megson@reading.ac.uk

S. Rajopadhye T. Risset
COSI, IRISA/INRIA,
Campus de beaulieu,
35042 Rennes cedex, France
{rajopadh,risset}@irisa.fr

Abstract

This paper is concerned with the uniformization of a system of affine recurrence equations. This transformation is used in the design (or compilation) of highly parallel embedded systems (VLSI systolic arrays, signal processing filters, etc.). In this paper, we present and implement an automatic system to achieve uniformization of systems of affine recurrence equations. We unify the results from many earlier papers, develop some theoretical extensions, and then propose effective uniformization algorithms. Our results can be used in any high level synthesis tool based on polyhedral representation of nested loop computations

1 Introduction

The advent of high density and high performance FPGA's such as the Virtex series from Xilinx, and also the trends towards increasing levels of VLSI integration (see for example, the predictions in the SIA roadmap[1]) has given a new impetus to the direct silicon compilation of compute-bound applications. This is evidenced by the relative maturity achieved by academic research systems such as MMAlpha [4], Compaan [2], etc., as well as the development of industrial prototypes such as HP lab's PICO [5] system. These systems are in effect, programming and/or rapid prototyping tools and systems in the context of highly parallel systems-on-chips. The target (virtual) architecture is usually in the form of a systolic array: a locally interconnected grid of simple processors with hardwired control (i.e. processing elements, PEs). FPGA's and other reconfigurable computing devices serve as an ideal (physical) target architecture for such PE arrays.

[1] `http://public.itrs.net/`

The well known methods of space-time mapping [15] can be very effectively used to derive such arrays, provided the applications are *perfectly nested* loops with *hyper rectangular iteration spaces* and *uniform (i.e., constant) flow dependences* (we will see that such computations are described by *Systems of Uniform Recurrence Equations* or SURE's that admit a linear schedule). However, many compute bound application codes are more general than this. They contain *imperfectly* nested loops with *affine dependences*, over *polyhedral iteration spaces* (modeled by *Systems of Affine Recurrence Equations* or SARE's). Such programs can be implemented either directly on silicon or compiled to a parallel general purpose machine by using the methods of a mathematical framework called polyhedral model [10, 24, 18, 7, 6]. However, systolic arrays remain a very efficient implementation alternative, for those (parts of) applications that are amenable to such an implementation. Hence it is useful and important to determine when and how to systematically transform a more general program specification into a form that is guaranteed to admit a systolic array implementation.

Methods to cast a SARE to a SURE have indeed been the subject of study by many authors [8, 18, 3, 16, 25, 21] and two main transformations, namely *pipelining* and *routing*, have been proposed. These two transformations have been incorporated in the MMAlpha design environment [4]. While these transformations can aid the designer with automatic rewriting of specification, the designer is left with the following tasks:

- choosing the appropriate transformation i.e., to decide when to apply *pipelining* or *routing*;
- computing the "direction" of the pipeline as spec-

ified by the so called uniformization vectors;

- ensuring that the design is consistent after conversion i.e., it is possible to schedule and map the resulting SURE.

In this paper, we present and implement a *complete* and *automatic* system to achieve uniformization. Specifically, we first unify the results in many of the earlier papers (notably [13, 20, 3, 16, 18, 21]), develop some theoretical extensions, and propose effective uniformization algorithms. We have also implemented our algorithms and integrated them into the MMAlpha system, which is freely available under the GNU public license.

The paper is organized as follows: in section 2 we state the basic notations that we use. The uniformization tool is presented in section 3. Section 4 reviews the work done on the uniformization of SAREs and we conclude in section 5

2 SARE: Specification and Properties

In regular array synthesis, algorithms are typically specified as a system of affine recurrence equations. Such a system comprises of a set of equations of the following form:

$$z \in \mathcal{IS} : U[z] = \mathcal{F}(\ldots, V[d(z)], \ldots) \qquad (1)$$

$\mathcal{IS}$, the iteration space, is the set of integral points in a *domain* (domains are usually convex polyhedra or unions of convex polyhedra) of $\mathbb{Z}^n$ (n is the dimension of U) and d is an affine function of z. These equations recursively define the values of the variables $(U, V, \ldots)$. Synthesizing regular arrays from such specifications involves a number of transformations. For a specification to be formally represented and manipulated, it should be expressible in a specific language. The *Alpha* programming language and its associated synthesis environment *MMAlpha* provides a framework which accounts for such requirements [4]. In what follows, the *Alpha* language constructs are used to specify systems of equations.

To apply standard synthesis transformations, recurrence equations are first transformed into *fully indexed* form [20]. In this form each variable (except input/output variables) has the same number of indices. By suitable pre-processing, a system of affine recurrence equations can always be transformed into this form [20], but remember, neither the preprocessing nor the resulting fully indexed SARE is unique (this preprocessing is close to the alignment problem [11, 1]). Although alignment can affect the final design, we do not study it in this paper (see [11] for solutions to that problem). From now on, when we refer to a SARE we will assume a fully indexed SARE.

2.1 Uniformity

An SARE implicitly defines data dependences. Each dependence involves the result of an equation and one of its arguments, denoted by: $U[z] \longmapsto V[d(z)]$, and read as "U at z depends on V at $d(z)$". A data dependence relation is established as follows: for each $z \in \mathcal{IS}$, the computation of $U[z]$ depends on the value of variable V computed at point $d(z)$ of the index space $\mathcal{IS}$ and hence the value $V[d(z)]$ should be available for the computation of $U[z]$ to be feasible. In the case of fully indexed SAREs, the data dependence relation is captured with the notion of *data dependence vectors*. Uniformization aims at converting all *non-uniform* dependences (as defined below) into *uniform* dependences.

Definition 2.1 *[Data dependence] : Consider a fully indexed recurrence equation*

$$z \in \mathcal{IS} \subset \mathbb{Z}^n : U[z] = \mathcal{F}(\ldots, V[d(z)], \ldots)$$

where $d(z) = A \cdot z + b, A \in \mathbb{Z}^{n \times n}, b \in \mathbb{Z}^n$. *Since* $U[z]$ *is computed using* $V[d(z)]$ *there is a direct (data) dependence from* U *at the point* z *to* V *at point* $d(z)$. *the value* $\Theta_{[A,b]}(z) = z - d(z) = (I_n - A) \cdot z - b$ *is defined.* $\Theta_{[A,b]}$ *is called* data dependence function *or simply* data dependence. *A data dependence is* uniform *if it is constant:* $\Theta_{[A,b]}(z)$ *does not depend on* z *i.e.,* $\forall z \in \mathbb{Z}^n$, $\Theta_{[A,b]}(z) = c$, *where* $c \in \mathbb{Z}^n$ *is a constant integer vector).*

Example 2.2 represents a fully indexed Alpha program for the Gauss-Jordan decomposition. The dependences and the corresponding dependence functions for example 2.2 are shown in table 1. The data dependence $A[i, j, k] \longmapsto A[i, j, k-1]$ over all the sub-domains is uniform. The rest of the data dependences are non-uniform. Notice that only the dependences between local (hence, fully indexed) variables are considered.

In this paper, we use a weaker condition for "*uniformity* " (as defined in [20]) so that complementary technique can be included in the uniformization tool (it can be seen as uniformity *in context*) This weaker condition takes into account the domain over which the data dependence is defined as follows:

line	Data dependence	$\Theta_{[A,b]}(i,j,k)$
18	A[i,j,k] $\longmapsto$ A[i,j,k-1]	(0,0,1)
18	A[i,j,k] $\longmapsto$ A[k,k,k-1]	(i-k,j-k,1)
19	A[i,j,k] $\longmapsto$ A[i,j,k-1]	(0,0,1)
20	A[i,j,k] $\longmapsto$ A[i,k,k-1]	(0,j-k,1)
20	A[i,j,k] $\longmapsto$ A[k,k,k-1]	(i-k,j-k,1)
20	A[i,j,k] $\longmapsto$ A[k,j,k-1]	(i-k,0,1)

Table 1. Dependences and dependence functions generated by the first two case branches in the definition of A in Example 2.2

Example 2.2 *:*

```
1  system MultiStepGaussJordan :   ({N | 4 ≤ N})
2    (a : {i, j | 1 ≤ i ≤ N; 1 ≤ j ≤ N} of integer;
3     b : {i | 1 ≤ i ≤ N} of integer )
4    returns (x : {i | 1 ≤ i ≤ N}of integer );
5  var
6     A : {i, j, k | k ≤ i ≤ k + N − 1; k ≤ j ≤ N + 1;
7                  1 ≤ k ≤ N} of integer ;
8     A_in : {i, j, k | 1 ≤ i ≤ N; 1 ≤ j ≤ N + 1; k = 0};
9  let
10    A_in[i, j, k] =case
12                 { | j ≤ N} : a[i, j];
13                 { | j = N + 1} : b[i];
14                 esac
15    A[i, j, k] =
17      case
18      { | i = 1; k = 1} : A_in[i, j, k − 1]/A_in[k, k, k − 1];
19      { | 2 ≤ i; k = 1; } : A_in[i, j, k − 1]−
20      A_in[i, k, k − 1]/A_in[k, k, k − 1]* A_in[k, j, k − 1];
21      { | N + 1 ≤ i ≤ k + N − 2} : A[i, j, k − 1]−
22      A[i, k, k − 1]/A[k, k, k − 1] ∗ A[k, j, k − 1];
23      { | i = k; 2 ≤ k} : A[i, j, k − 1]/A[k, k, k − 1];
24      { | k + 1 ≤ i ≤ N; 2 ≤ k} : A[i, j, k − 1]−
25      A[i, k, k − 1]/A[k, k, k − 1] ∗ A[k, j, k − 1];
26      { | i = k + N − 1; 2 ≤ k} : A[i − N, j, k − 1]−
27      A[i − N, k, k − 1]/A[k, k, k − 1] ∗ A[k, j, k − 1];
28      esac
29    x[i] =case
30          { | 1 ≤ i ≤ N − 1} : A[i + N, N + 1, N];
31          { | i = N} : A[i, N + 1, N];
32          esac
33 tel;
```

Definition 2.3 *[Uniform Data Dependence over a domain $\mathcal{IS}$] : A dependence $U[z] \longmapsto V[Az+b]$ is uniform over a sub-domain $\mathcal{IS} \subseteq \mathbb{Z}^n$ if there exists a constant integer vector $c \in \mathbb{Z}^n$ such that $\forall\, z \in \mathcal{IS}\ \ \Theta_{[A,b]}(z) = c$.*

Observe that uniformity may not be apparent from (A, b) alone. For example, the dependence $A[i, j, k] \rightarrow A[k, j, k-1]$ is apparently non-uniform, however if it occurs on a sub-domain where $i = k$ then the dependence vector is constant $(0, 0, 1)$ and the dependence is uniform. This notion of uniformity is "local" with respect to an equation domain $\mathcal{IS}$ and and not over the whole space $\mathbb{Z}^n$. Data dependences which are "globally" uniform over $\mathbb{Z}^n$ are, however, uniform over each subset of $\mathbb{Z}^n$. A data dependence with local uniformity has the following property (the proof can be found in [20]):

Property 2.4 *[Local Uniformity] : A data dependence $\Theta_{[A,b]}$ is uniform over a domain $\mathcal{IS} \subseteq \mathbb{Z}^n$ if and only if[2] $lin(\mathcal{IS}) \subseteq null(I_n - A)$.*

This property can be exploited to obtain certain simplifications during the uniformization process. Data dependences with local uniformity can be converted into uniform dependences through a simple renaming transformation. After renaming, the dependence function $\Theta_{[A,b]}(z)$ will be of the form $\Theta_{[A,b]}(z) = c$.

2.2 Compatibility

Uniformization substitutes affine data dependences of a SARE with a new set of uniform data dependences. An important issue to be considered is whether these new dependences are "compatible" with existing dependences. We consider a set of dependences to be "compatible" if the corresponding SURE can be scheduled with an affine schedule. The *scheduling* problem is to find a function, called *timing function*, which associates a time instant to each variable instance $U[z], z \in \mathcal{IS}$. We are interested in *affine timing functions*:

Definition 2.5 *[Affine Timing Function] [16]: an affine timing function is a set of mappings $t_U : \mathbb{Z}^n \rightarrow \mathbb{N}$ (for each variable U of the SARE) of the form $t_U(z) = \lambda^t \cdot z + \alpha_U$, where $\lambda \in \mathbb{Z}^n, \alpha_U \in \mathbb{Z}$ such that, if the computation of variable instance $U[z]$ depends on the instance $V[d(z)]$; then $t_U(z) > t_V(d(z))$).*

Note that, in the previous definition λ (the linear part) is common to all variables while α (the constant part) may be distinct for each variable (these types of timing functions are often called *shifted linear schedules*). To ensure compatibility, we still need the notion of *data dependence cones*. A dependence cone $\Theta^*_{[A,b]}$ corresponding to a dependence function $\Theta_{[A,b]}$ is defined as follows:

Definition 2.6 *[Data dependence cone] : For a given dependence function $\Theta_{[A,b]}$ existing on domain $\mathcal{IS}$,*

[2] $lin(\mathcal{IS})$ denotes the lineality space of domain $\mathcal{IS}$, i.e. the smallest affine space containing $\mathcal{IS}$; $null(A)$ denotes the null space of function A.

the data dependence cone, denoted $\Theta^*_{[A,b]}$, is defined as the smallest polyhedral cone[3] *containing its range $\Omega_{[A,b]}$ i.e., $\Theta^*_{[A,b]} = cone(\Omega_{[A,b]})$, where $\Omega_{[A,b]} = \Theta_{[A,b]}(\mathcal{IS})$.*

For the example 2.2, the range corresponding to the dependence function $\Theta_{[A,b]} = (i,j,k \rightarrow 0, j-k, 1)$, (line 20 of example 2.2) is $\Omega_{[A,b]} = \{i,j,k \mid i = 0;\ k = 1, j \geq 0\}$ and its dependence cone is $\Theta^*_{[A,b]} = \{i,j,k \mid i = 0;\ k \geq 0, j \geq 0\}$.

Property 2.7 *[Timing Function and Dependence Cones] : the variables in a system of recurrence equations admit a common affine timing function (or can be scheduled) if the set of its dependence vectors form a pointed cone*[4] *[16, 20]. This means that the dependence cone Θ^* should not have two rays, ϵ_1, ϵ_2 such that $\epsilon_1 = -\epsilon_2$.*

Hence, the "pointedness" of the dependence cones is a sufficient condition for schedulability. It may be easily checked that it is also a necessary condition because our affine timing functions all have the same linear part λ. During our uniformization process, we will ensure at each step that this property remains invariant.

3 Uniformization Tool

Converting SAREs to SUREs implies that all dependence vectors should be transformed to the form $\Theta_{[A,b]} = c$. This is achieved by decomposing the dependence vectors $\Theta_{[A,b]}(z)$ into parts ($\Theta_{[A,b]}(z) = \sum_{j=1}^{q} \alpha_j(z) \cdot \phi_j$) in which the ϕ_js (called *uniformization vectors*), satisfy particular criteria and the $\alpha_j(z)$ are scalar functions of z. The uniformization vectors can be used either for *pipelining* or *routing* the dependence. We first explain in detail the kernel transformations implemented in MMAlpha: pipelining, routing and local uniformizing. Then we briefly explain the overall algorithm of the uniformization process.

3.1 Pipelining

This most well known transformation was introduced independently by Quinton & Van Dongen [3] and Wong & Delosme [25] and Rajopadhye et al. [18] (early ideas were present in [8], see also [26]). It transforms a broadcast into the propagation of a value. Consider an equation:

$$z \in \mathcal{IS}, \quad U[z] = \mathcal{F}(\ldots, V[Az+b], \ldots), \qquad (2)$$

[3]Polyhedral cones are a special case of polyhedra with only a single vertex (which can be treated as the origin) [22]. Hence they can be described with only extremal rays.

[4]A cone C is pointed if there exists a vector π such that $\forall x \in C, \pi \cdot x > 0$.

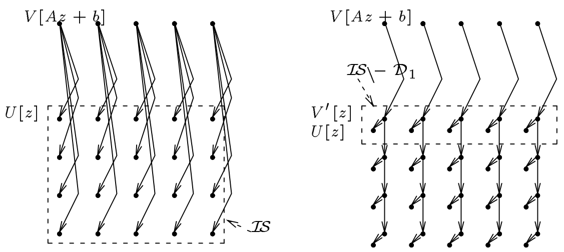

Figure 1. Example of pipeline Transformation, the arrows represent the data flow (opposite to the dependence)

such that the dependence $U[z] \longmapsto V[Az+b]$ is non uniform and that we have: $lin(\mathcal{IS}) \cap null(A) \neq \emptyset$. Then, for any vector $v \in lin(\mathcal{IS}) \cap null(A)$, we can write an equivalent SARE (see figure 1):

$$z \in \mathcal{IS}, \quad U[z] = \mathcal{F}(\ldots V'[z] \ldots) \qquad (3)$$

$$z \in \mathcal{D}_1, \quad V'[z] = V'[z-v] \qquad (4)$$

$$z \in \mathcal{IS} \setminus \mathcal{D}_1, \quad V'[z] = V[Az+b] \qquad (5)$$

where $\mathcal{D}_1 = \mathcal{IS} \cap (\mathcal{IS} + v)$, ($\mathcal{D}_1$ is the intersection of $\mathcal{IS}$ with itself shifted by v, i.e., the set of points $z' \in \mathcal{IS}$ such that $(z'-v)$ also belongs to $\mathcal{IS}$). We prove hereafter that value of U is exactly the same in the new program.

It is obvious that $\mathcal{D}_1 \cup (\mathcal{IS} \setminus \mathcal{D}_1) = \mathcal{IS}$ and that $\mathcal{D}_1 \cap (\mathcal{IS} \setminus \mathcal{D}_1) = \emptyset$ hence the variable V' is defined on all the domain $\mathcal{IS}$. Let z be a point of $\mathcal{IS}$, we consider the set $\mathcal{E} = \{z - iv \mid i \in \mathbb{N}, z - iv \in \mathcal{IS}\}$. We can show by induction that, provided that the set $\mathcal{E}$ is not infinite along v, there exists $i_0 \in \mathbb{N}$ such that $z - iv \in \mathcal{D}_1,\ \forall i \leq i_0$ and $z - (i_0+1)v \in \mathcal{IS} \setminus \mathcal{D}_1$. In that case, we have $V'[z] = V'[z-(i_0+1)v] = v[A(z-(i_0+1)v)+b] = V[Az+b]$ (because $v \in null(A)$), Hence, $U[z]$ is set to the same value as in equation (2).

This proof can be found in [3, 18]. It is interesting to note that we have a lot of freedom for uniformizing the dependence. The direction of the pipelining vector will be guided by the compatibility conditions mentioned in section 2.2 and by a possible infinite direction in the domain. The vector v does not have to be unitary (GCD of its components is 1), but in general this leads to simpler control in the resulting architecture. The only difficulty in the pipelining process lies in the choice of the pipelining vectors. Quinton and Van Dongen [3] proved that, provided that the dependence cone was pointed before the pipelining process, it was always possible to chose pipelining vectors that keep that dependence cone pointed. In practice, all the

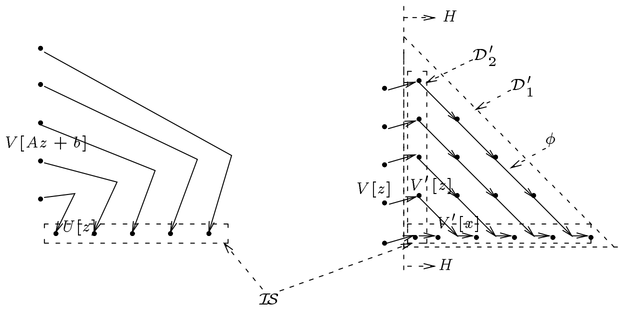

Figure 2. Example of routing proposed in equations (7,8,9),the arrows represent the data flow (opposite to the dependence)

computation required in order to rewrite equation (2) into equations (3-5) can be done with elementary manipulation of unions polyhedra (intersection, image by a one to one mapping, difference).

3.2 Routing

Pipelining is not always possible. If $\Theta_{[A,b]}$ is not uniform in the context of $\mathcal{IS}$ and if the condition $lin(\mathcal{IS}) \cap null(A) \neq \emptyset$ is not met, pipelining is not possible and in such cases we will need to apply the *routing* transformation. The routing process is similar to the pipelining process, but the routing vectors do not belong to $lin(IS)$. Finding routing vectors consists in finding an integral decomposition of the dependence $\Theta_{[A,b]}(z) = \sum_{j=1}^{q} \alpha_j(z) \cdot \phi_j$ where the $\alpha_j(z)$ are scalar affine functions and none of the ϕ_j belongs to $lin(\mathcal{IS})$ [3, 20]. Indeed, if a routing vector is in $lin(\mathcal{IS})$, we will have to use additional channels to transfer these values so as to avoid conflict with existing transfer [21].

In [3], Quinton and Van Dongen proposed a way to solve the problem: they gave a way to find the ϕ_js and the above decomposition. This may not be possible without increasing the dimension of the space if $dim(lin(\mathcal{IS})) = n$. Once the decomposition $\Theta_{[A,b]}(z) = \sum_{j=1}^{q} \alpha_j(z) \cdot \phi_j$ is done, we treat each ϕ_j successively by the kernel routing transformation below (each time, also verifying that the ϕ_j is compatible with the scheduling cone).

Formally the kernel routing transformation is expressed as follows, given an equation:

$$z \in \mathcal{IS}, \quad U[z] = \mathcal{F}(\dots V[Az+b] \dots) \tag{6}$$

where the (non-uniform) dependence $\Theta_{[A,b]}(z)$ can be written as: $\Theta_{[A,b]}(z) = (I_n - A) \cdot z - b = \alpha(z) \cdot \phi$ and $\phi \notin lin(\mathcal{IS})$. Then, we can write an equivalent SARE:

$$z \in \mathcal{IS}, \quad U[z] = \mathcal{F}(\dots V'[z] \dots) \tag{7}$$

$$z \in \mathcal{D}'_1, \quad V'[z] = V'[z-\phi] \tag{8}$$

$$z \in \mathcal{D}'_2, \quad V'[z] = V[z] \tag{9}$$

Where $\mathcal{D}'_1 = \{z - k\phi \mid z \in \mathcal{IS};\ 0 \leq k < \alpha(z)\}$ and $\mathcal{D}'_2 = \{z - \alpha(z)\phi \mid z \in \mathcal{IS}\}$ ($\mathcal{D}'_1$ and $\mathcal{D}'_2$ may not be polyhedra, we explain how to compute them later). This new specification is equivalent to the previous one because the function $z \mapsto z - \alpha(z)\phi = Az + b$ is linear and one to one on $\mathcal{IS}$ (as $\phi \notin lin(\mathcal{IS})$).

From the implementation point of view, the computation of the domains $\mathcal{D}'_1$ and $\mathcal{D}'_2$ is a little more complicated because they are not defined using usual operators on polyhedra. We propose a way to build them. We first add an infinite direction to $\mathcal{IS}$ in the direction of $-\phi$, we obtain an infinite polyhedron in which the routing will take place. The space H where the routing ends can be computed by taking the image of $\mathcal{IS}$ by the function ($z \mapsto z - \alpha(z)\phi$) which is linear and invertible (hence the image is included in a polyhedron). We know that ϕ is not in this space, hence H 'cuts' the extension of $\mathcal{IS}$ that we have just built. By intersecting these two domains we get the exact domain of routing (H is represented on figure 2).

Notice that, if the dot product of ϕ and the normal to the hyper-plane defined by H is not 1 (or -1), then the domain on which the routing ends may not be a polyhedron (it will be a so called $\mathbb{Z}$-polyhedron, with regular holes in it), we are not able to deal with this cases in the current version of MMAlpha, however, the polyhedral library has been extended to $\mathbb{Z}$-polyhedra [17] and an extension to handle this is being developed.

There remains the problem of computing the decomposition $\Theta_{[A,b]}(z) = \sum_{j=1}^{q} \alpha_j(z) \cdot \phi_j$. We use an unimodular extension of the dependence cone (as proposed in [3]). One way to do so is the following:

- get a basis $B = (b_1, \dots, b_k)$ of the k-dimensional space spanned by the dependence cone (choose integral vectors).
- Deduce from this basis a Hermite basis (basis of $\mathbb{Z}^n$) whose k first vectors span this space (using the Hermite decomposition: $B = HU$, H being lower triangular, only the first k columns of U are used for computing the k columns of B)
- Use a symbolic solver (we used the `LinearSolve` function of Mathematica) to solve the linear system: $(u_1, \dots, u_k) \cdot (x_1, \dots, x_k)^T = \Theta_{[A,b]}(z)$, where z is symbolic. In the solution the x's are expressed in terms of z.

- If some vectors u_j need to be decomposed, then decompose them using a vector not in the current space.

In [3, 20], it was proposed that the decomposition vectors should not be in $lin(\mathcal{IS})$ for the routing transformation. This is not exactly the correct condition to check. In [12] we give examples where this condition ($\phi_j \notin lin(\mathcal{IS})$)) is not respected and the routing is valid or where this condition is respected but where the routing is impossible. In practice, the order in which the ϕ_j's are used during successive routing steps is important. Hence the following question arises: given a decomposition of $\Theta_{[A,b]}(z)$, does there exists an ordering of the decomposition vectors such that the routing can be performed? This is still an open question.

3.3 Auxiliary Transformations

In addition to the two main techniques discusses above, namely pipelining and routing, two auxiliary transformations were introduced in [20]. The first transformation *LocalUniformity* is applied before attempting the *pipelining* transformation whereas the second transformation *InputAlignment* is applied before the routing transformation.

3.3.1 Local uniformity

The *Local Uniformity* transformation uses the property (2.4) to track "apparent" non-uniformities during uniformization. A technique for testing the *LocalUniformity* condition was presented in [20] but the method to rewrite the non-uniform dependence as a uniform dependence was not solved. Here we present a way in which this rewriting can be accomplished.

When a local uniformity is identified, it implies that the index mapping $d(z) = Az + a$ can be substituted by an equivalent "uniform" index mapping given by $I_n \cdot z + c$. The constant vector c has to be determined to perform this re-indexing. It is obtained by eliminating variables from the equation $(A - I_n) \cdot z - c = -b$ using the set of implicit equalities which occur in the domain: $\mathcal{IS} = \{z \mid Bz = C, B'z \leq C'\}$. Hence, this amounts to solving the following matrix equation:

$$\begin{bmatrix} (A - I_n) & -I_n \\ B & 0 \end{bmatrix} \begin{bmatrix} z \\ c \end{bmatrix} = \begin{bmatrix} -b \\ C \end{bmatrix}$$

3.3.2 Input Alignment

This transformation can be used before routing transformation is attempted. The main idea in eliminating non-uniformity is to define a new input equation instead of introducing routing equations. More formally, given an equation:

$$z \in \mathcal{IS}, \quad U[z] = \mathcal{F}(\ldots, V[Az + b], \ldots) \qquad (10)$$

in which $U[z] \longmapsto V[Az + b]$ is non-uniform and an input equation:

$$z' \in \mathcal{IS}', \quad V[z'] = v[A'z' + b'] \qquad (11)$$

where $A' \in \mathbb{Z}^{l \times n}, b' \in \mathbb{Z}^l, l \leq n$. If the condition $\forall\, z \in \mathcal{IS}, Az + b \in \mathcal{IS}^{'}$ is satisfied then it is possible to rewrite the dependence by introducing a single input equation instead of the routing equations as in equations (

where the first two equations are as before and the third equation is the new input equation with $A'' = A' \circ A$ and $b'' = A'b + b'$. The new input equation re-defines the inputs to the system without altering the computations of the system. Thus the affine time function remain unchanged. The resulting system is uniform as the non-uniformities appear only in the input equations. Essentially, this transformation localizes the input values to a system and as a result the non-uniformities get eliminated.

All computations required in rewriting the non-uniform dependence, such as checking the condition $\forall\, z \in \mathcal{IS}, Az + b \in \mathcal{IS}^{'}$ and the indexing function $A''z' + b''$, can be carried out using routines from the polyhedral library in *MMAlpha*.

As an example, consider the following system of equations:

$$\begin{array}{r} (i,j) \in \mathcal{IS} = \{i,j \mid j = 0;\ 1 \leq i \leq 4\}, \ \ X[i,j] = X[0,i] \\ (i,j) \in \mathcal{IS}' = \{i,j \mid i = 0;\ 1 \leq j \leq 4\}, \ \ X[i,j] = x[j] \end{array}$$

The data dependences of the two equations are given by,

$$A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, A' = \begin{bmatrix} 0 & 1 \end{bmatrix}, b' = \begin{bmatrix} 0 \end{bmatrix}.$$

The affine transformation $[A, b](\mathcal{IS}) = \mathcal{IS}'$ and therefore the system can be rewritten as:

$$\begin{array}{r} z \in \mathcal{IS} = \{i,j \mid j = 0;\ 1 \leq i \leq 4\}, \ \ X[i,j] = X''[i,j] \\ z^{'} \in \mathcal{IS}^{'} = \{i,j \mid i = 0;\ 1 \leq j \leq 4\}, \ \ X[i,j] = x[j] \\ z \in \mathcal{IS} = \{i,j \mid j = 0;\ 1 \leq i \leq 4\}, \ \ X''[i,j] = x[i] \end{array}$$

The resulting system is uniform as the non-uniformities are only in the input equations.

3.4 Algorithm for Uniformization

The overall process of uniformizing a given SARE is summarized in procedure *UniformizeSystem*. The procedure takes as input an SARE and returns an SURE. It iterates over the equation set, transforming each equation with non-uniform dependencies into an equation in which all dependences are uniform. In this transformation process additional equations are introduced and concatenated to the original set of equations. For equation E_i with non-uniform dependences, the procedure *UniformizeDependence* constructs a set of uniform equations E_i^u. The process involves substituting variables in E_i (a substituted equation is represented as $E_i^{'}$) with new variables and introducing a new set of (uniform) equations. These new set of equations perform a simple data copy operation.

```
Procedure UniformizeSystem(S : SARE)
   begin
     U = ∅
     while ∃ non-uniform deps
       If ∃ non-uniform deps in eq E_i
          for each non-uniform deps in eq E_i
              E_i^u = UniformizeDependence(DD)
              U = U ⋀ E_i' ⋀ E_i^u
       else
           U= U ⋀ E_i
     end while
     Return{U}
   end

Procedure UniformizeDependence(DD)
   1:  Try LocalUniformity(DD)
   2:  else try InputAlignment(DD)
   3:  else try Pipelining(DD)
   4:  else apply Routing(DD)
   Return{new system}
```

For a particular dependence, the order in which the techniques should be applied is as shown in the procedure *UniformizeDependence*. The dependence can be pipelined when $lin(\mathcal{IS}) \cap null(A) \neq \emptyset$, the condition for meeting input alignment or local uniformity have been explained previously. If none of these cases hold, routing is possible if $\mathcal{IS} \setminus null(A) \neq \emptyset$ (because $lin(\mathcal{IS}) \cap null(A) = \emptyset$). The method presented here does not guarantee uniformization of *one to one* dependences occurring on full dimensional domains. The reason for this order of transformation is the following: routing introduces more resources (buffers) in the resulting array design compared with pipelining whereas local uniformity transformation does not introduce any resources at all.

As explained above, it is important to derive compatible dependences so that the resulting SURE can be scheduled. To do so, we have to verify that the "pointedness" condition (property 2.7) is met when each uniformization technique is applied.

We first verify the"pointedness" of the dependence cone of the original system Θ^*. Then, in order for the transformed system to be consistent it is necessary and sufficient that the dependence cone Θ^* and the chosen set of uniformization vectors $(\{\Phi\})$ should form a pointed cone. Therefore, for each technique this criterion will be applied in the selection of the uniformization vectors $\{\Phi\}$. For *pipelining* and *routing* we have chosen the extremal rays of the dependence cone $\Theta^*_{[A,b]}$ as the uniformization vectors. This ensures the condition for "pointedness" of dependence cones and thereby the existence of a timing function for the transformed system. For the *local uniformity* case the index mapping functions are directly modified. This modification does not affect the consistency of the system of equations and hence the condition for existence of timing function is preserved.

4 Related work

The topic of converting non-uniform specifications into an equivalent uniform specification has received considerable attention in the last decade [8, 26, 18, 16, 25, 21, 14]. The interest stems because, to allow more abstract high level specifications, people have to link up with the classical synthesis design steps which are based on SUREs as the starting specification. All proposed techniques share the same basic intuition of routing data in a uniform manner through the computation domains of the system of recurrence equations. They, however, differ in the class of recurrences considered and the set of requirements that are imposed in the technique.

The earliest work dealt with the class of affine recurrence equations, amongst these proposals only [16] considered the issue of *compatibility* (section 2.2).

A more general uniformization framework was presented by Roychowdhury in [21]. He considers the uniformization of recurrences in which the indexing function of both the l.h.s and r.h.s variables in equation (1) are affine functions of index vector z i.e., the equations are of the form:

$$z \in \mathcal{IS} : U[d(z)] = \mathcal{F}(\dots, V[d(z)], \dots)$$

However, his uniformization is achieved without any reference to compatibility. Hence, there is no guarantee that the resulting system can be scheduled. Indeed, his routing scheme (called *multidimensional propagation*) often uses uniformization vectors which will result in a uniform system requiring piece-wise affine schedules as opposed to strict affine schedules. In his scheme he uses the *idempotency* of indexing function as a condition for applying pipeline transformation. If idempotency is not satisfied then he applies routing. As the following example demonstrates, idempotency is not necessarily the strict condition to be used to demarcate between pipelining and routing. It is possible to avoid routing even when idempotency is not satisfied, through judicious input alignment.

Consider the following system of equations:

$$\begin{aligned} z \in \mathcal{IS} = \{i,j \mid 1 \le i,j \le N\}, &\ \ Y[i,j] = X[0,i] \\ z' \in \mathcal{IS}' = \{i,j \mid i = 0;\ 1 \le j \le N\}, &\ \ X[i,j] = x[j] \end{aligned}$$

where the second equation is an input equation. The indexing function for the non-uniform dependence $Y[i,j] \longmapsto X[0,i]$ is specified by,

$$A = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

It is clear that the indexing function is not idempotent. However, we can still pipeline the dependence using the null space vector $(0,1)$ as follows:

$$\begin{aligned} z \in \mathcal{IS} = \{i,j \mid 1 \le i,j \le N\}, &\ \ Y[i,j] = X'[i,j] \\ z' \in \mathcal{IS}' = \{i,j \mid 1 \le i,j \le N\}, &\ \ X'[i,j] = X'[i,j-1] \\ z' \in \mathcal{IS}'' = \{i,j \mid j = 0;\ 1 \le i \le N\}, &\ \ X'[i,j] = X[0,i] \end{aligned}$$

The non-uniformity is now confined to the pipeline initialization equation (last equation above). This can, however, be made uniform with *InputAlignment* transformation.

More recently, uniformization of so called *Integral* and *Dynamic* recurrence equations, which extends beyond affine specifications, have been introduced in [13, 19]. Their uniformization scheme enables the systematic derivation of regular array architectures for algorithms with non-affine specifications such as the *k-napsack* problem. Compatibility is ensured with this technique as affine schedules are preserved by their uniformization scheme [19]. However, this technique cannot be easily integrated as the automation of this technique requires a design environment that is capable of processing specifications in which both the index functions and the domains are non-affine.

Although a broad set of techniques dealing with a range of specifications have been studied, their integration into design flows has been limited. Pipelining and routing transformations have been previously implemented in MMalpha [4], however the computation of pipeline/routing vectors was left to the designer. Also, compatibility issue was not addressed. Similarly, the SADE system [9] also provided pipelining and routing transformations compatibility. Both these issues are dealt with in our implementation. In addition, we have also integrated auxiliary transformations which can provide improved uniformization results (examples obtained after using the tool are given in [12]).

From the theoretical point of view, several questions remain open. For instance, the global optimality for a uniformization procedure cannot be reached currently. Defining a good cost function ([23, 1]) for a uniformization transformation is a key issue for that problem. Shang et all [23], also address the uniformization of affine dependence programs. Their key contribution consists of a number of heuristic techniques that collectively seek to optimize the schedule of the resulting program. The authors do not address the problem of program transformation and systematic derivation of the resulting uniform program. Moreover, optimality of the linear schedule is only one of the cost measures. Nevertheless, their methods are complementary to ours, and could be implemented in our tool to guide to choice of the uniformization vectors.

5 Conclusion

In this paper, we have unified some of the techniques developed for the case of localizing affine recurrence equations and also highlighted some of the problems that need to be overcome in the routing transformation. We have also demonstrated how auxiliary transformations can be used to enhance a uniformization framework by avoiding pipelining and routing under certain conditions. Although considerable effort has gone into the theory of uniformization, it was not previously matched by implementation. Our objective in realizing a implementation of well known techniques is to make the available theory effectively usable so that designers can apply these techniques without engaging in detailed consideration of the underlying mathematics required in the conversion process. The main difficulties of applying uniformization techniques such as choosing appropriate transformation, synthesizing uniformization vectors and ensuring that the uniformization is carried out in a consistent way, is overcome in our tool.

The primary aim of this tool is to simplify the early stages of the design process of systolic array synthesis by automatic production of SUREs. The impact of regularization of the qualities of the target design, such as space-time optimality, is not considered. Therefore designers may have to iterate the procedure with different choices of uniformization vectors to explore the alternatives. For any valid choice, however, the uniformization tool will deliver a system that can be scheduled. Some optimizations through additional techniques (*localUniformity* or *inputAlignment*) that we use, is achieved in our uniformization scheme. Other objective functions, proposed in [23], may be used to improve the results of uniformization.

References

[1] K. Danckaert. *Loop Transformations for Data Transfer and Storage Reduction on Multiprocessor Systems.* PhD thesis, Katholoeke Universiteit Leuven, 2001.

[2] E. Deprettere, E. R. a nd P. Lieverse, and B. Kienhuis. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, May 2000.

[3] V. V. Dongen and P. Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays. In K. B. et al. eds., editor, *International Conference on Systolic Arrays*, pages 473–482. IEEE Computer Society Press, 1988.

[4] F. Dupont de Dinechin, P. Quinton, S. Rajopadhye, and T. Risset. First Steps in Alpha. Technical Report 1244, Irisa, 1999.

[5] R. S. et al. High-Level Synthesis of Non Programmable Hardware Accelerators. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, Boston, July 2000.

[6] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, Oct. 1992.

[8] J. Fortes and D. Moldovan. Data broadcasting in linear schedule array processors. In *11'th Annual Symp. on Computer Architecture*, pages 224–231, 1984.

[9] G.M. Megson and D. Comish. Systolic algorithm design environments (SADEs). In *Transformational Approaches to Systolic Design*, chapter 9. London : Chapman and Hall, 1994.

[10] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[11] J. Li and M. Chen. The data alignent phase in compiling programs for distributed memory machines. *Journal Parallel Distributed Computing*, 13:213–221, 1991.

[12] M. Manjunathaiah, S. Rajopadhye, and T. Risset. Uniformization tool for systolic array designs. Technical Report 1350, Irisa, Rennes, 2000.

[13] G. Megson and L. Rapanotti. Uniformization techniques for reducible integral recurrence equations. In *Algorithms and Parallel VLSI Architectures III*, pages 283–296. Elsevier Science Publishers B.V., 1995.

[14] C. Mongenet. Affine dependence classification for communications minimization. *International Journal of Parallel Programming*, 25(6):497–524, 1997.

[15] P. Quinton. The systematic design of systolic arrays. In *Automata networks in Computer Science*, pages 229–260. Manchester University Press, 1987.

[16] P. Quinton and V. V. Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1:93–113, 1989.

[17] P. Quinton, S. V. Rajopadhye, and T. Risset. On manipulating Z-polyhedra using a canonical representation. *Parallel Processing Letters*, 7(2):181–194, June 1997.

[18] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241.

[19] L. Rapanotti. *On the Synthesis of Integral and Dynamic Recurrences.* PhD thesis, University of Newcastle upon Tyne, U.K., March 1996.

[20] L. Rapanotti and G.M.Megson. Pre-processing in SADE : Stage III. Technical Report 471, The University of Newcastle upon Tyne, Computing Science, U.K., March 1994.

[21] V. Roychowdhury. *Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms.* PhD thesis, Stanford University, Department of Electrical Engineering, Stanford, CA, December 1988.

[22] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley Interscience series in Discrete Mathematics and Optimisation, 1986.

[23] W. Shang, E. Hodzic, and Z. Chen. On uniformization of affine dependences algorithms. *IEEE Transactions on Computers*, 45(7):827–840, June 1996.

[24] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In *EUROPAR 96: Parallel Processing Conference*, pages 389–397, Lyon, France, August 1996. Springer Verlag.

[25] Y. Wong and J. Delosme. Transformation of broadcasts to propagations in systolic algorithms. *Journal of Parallel and Distributed Computing*, 14(2):121–145, 1992.

[26] Y. Yaacoby and P. R. Cappello. Converting affine recurrence equations to quasi-uniform recurrence equations. In *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory.* Springer Verlag, June 1988.