Mapping Linear Recurrences onto Systolic Arrays

Ladan Kazerouni  
Department of Mathematics  
University of Bombay  
Bombay, INDIA.  
ladan@irearn.bitnet

Basant Rajan and R.K. Shyamasundar  
Computer Science Group  
Tata Institute of Fundamental Research  
Bombay 400 005, INDIA  
{basant, shyam}@tcs.tifr.res.in

Abstract

We present an automatic method for mapping a system of linear recurrence equations onto systolic architectures. First, we show that systolic architectures can be derived from linear recurrence equations using the notion of directed recurrence equations. Next, we provide a procedure called CUBIZATION to achieve better performance while mapping such equations. The CUBIZATION procedure is completely automated and can be implemented to design systolic programs for a very general architecture referred to as Basic Systolic Architecture (BSA). Using BSA, we obtain specific target architectures. The method is illustrated using a running example of Gauss-Jordan diagonalization.

1 Introduction

In [6], we proposed a method to map a system of directed recurrence equations (DRE), a subclass of linear recurrence equations (LRE), which properly includes uniform recurrence equations onto systolic architecture. Here, we extend our method to LREs where the dependency vector at a point is a linear function of the point.

The problem of mapping LREs to systolic architectures has been addressed in the past in [8, 10]. However, their methods depend heavily on finding an affine timing function and required one to solve a system of linear programming problems [3, 8, 10]. Jagadish et al. [3], Quinton and Dongen [8] arrive at a valid schedule for some specific examples by studying the constraint matrix and altering the problem or by other ad-hoc methods. In contrast, our approach is capable of mapping LREs onto systolic architectures in an automatic fashion, even when finding an appropriate timing function might fail in other approaches.

In this paper, we develop a simple method for deriving systolic arrays from LREs. First, we show that every linear recurrence equation can be expressed as a system of directed recurrence equations (from where our method [6] for DREs is applicable). Next we give a more efficient procedure called CUBIZATION that transforms the dependency graph of the recurrence equations into a graph we term the Modified Dependency Graph (MDG) where a node represents an equivalent one under the constraint that the out degree of a node in the MDG must be bounded above by a constant independent of the domain size. Our approach is therefore scalable with respect to domain size.

2 Definitions and notations

Definition 1 (Linear recurrence equation) Let \( I_n \) denote the integer lattice in Euclidean \( n \)-space \( \mathbb{R}^n \). Any point \( p \in I_n \) can be designated by an \( n \)-dimensional vector whose coordinates are integers. Let \( L_{n_1} \) be a convex subset of \( I_n \). A system of linear recurrence equations (LRE) is a collection of equations of the following form:

\[
\begin{align*}
\delta_{i_1}(p) &= A_i p + b_i, \\
\delta_{i_2}(p) &= A_{i_2} p + b_{i_2},
\end{align*}
\]

where

1. \( a_i \) is a function from \( L_{n_1} \) to \( R \).
2. \( p \in L_{n_1} \) and \( v_{i_1} \) is an initial value in \( R \) given apriori for a boundary point \( p_{i_1} \in L_{n_1} \).
3. \( A_i \) is an affine function from \( L_{n_1} \) to \( L_{n_1} \) of the form

\[
\delta_i(p) = A_i p + b_i,
\]

where

(a) \( A_i \) is a constant \( n_1 \times n_1 \) matrix over \( Z \),
(b) \( b_i \) is a constant \( n_1 \times 1 \) matrix over \( Z \),
4. \( a_k(p) \) does not appear on the r.h.s. of the equation,
5. \( f_k \) is a function from \( R^m \) to \( R \).

Definition 2 (Normal linear recurrence equation) [8] A LRE is called normal if all the transformations \( \delta_i \) are from \( L_n \) to \( L_n \) for a fixed \( n \).

Definition 3 (Uniform recurrence equation) [4] A linear recurrence equation is said to be uniform if it is normal and the transformations \( \delta_i \) preserve differences. i.e. \( p - p' = \delta_{i_k}(p) - \delta_{i_k}(p') \) for \( 1 \leq i, k \leq s \).
This in turn would constrain $A_{ik}$ to be the identity matrix of order $n$.

**Definition 4** (Directed recurrence equation) [6] A normal linear recurrence of the form (1) is said to be directed, if $\forall i, k, b_{ik}$ is an affine transformation from $L_n$ to $L_n$ such that $\delta_{ik}(p) = A_{ik}(p + b_{ik})$ and $\dim(ker(A_{ik})) \leq 1$. When $\dim(ker(A_{ik})) = 0$, $b_{ik} = A^{-1}_{ik}b_{ik}$ and when $\dim(ker(A_{ik})) = 1$, $A_{ik}(b_{ik}^*) = 0$ and $b_{ik}^* \neq 0$. Here, $\dim(ker(A_{ik}))$ refers to the dimension of the kernel (or the null space) of $A_{ik}$.

### 3 DREs as a basic tool

In this section, we show that every linear recurrence can be expressed as a system of DREs.

**Lemma 1** Every normal linear recurrence equation can be expressed as a system of DRE.

**Proof:** Consider the following general system of normal linear recurrence equations.

Fix $s \in N$. For $k \in \{1, \ldots, s\}$,

\[
a_k(p_0^k, \ldots, p_n^k) = f_k(a_i(\delta_{ik}(p)), \ldots, a_s(\delta_{ik}(p)))
\]

(2)

where $p \in L_n$, $\delta_{ik}(p) = A_{ik} p + b_{ik}$, $A_{ik}$ is a constant $n \times n$ matrix and $b_{ik}$ is a constant $n \times 1$ matrix. Let $A_{ik} p + b_{ik} = p^0$, $p^0 = p$ and further let $p^b$ denote the $j^{th}$ component of $p^b$.

Then equation (3) can be written as:

\[
a_k(p_0^k, \ldots, p_n^k) = f_k(a_i(p_0^i, \ldots, p_n^i), \ldots, a_s(p_0^s, \ldots, p_n^s))
\]

(4)

where $a_i^1 = a_i$. Assume that $p^0$ and $p^1$ differ in $r$ components and let $m$ be the first position at which $p^0$ differs from $p^1$. Let $p^2$ be defined as $(\forall l \geq 0, t \leq m, p^2_l = p^1_l)$ and $(\forall l \geq m + 1, t \leq n, p^2_l = p^1_l)$. Then inductively $p^{t+1}$ is defined as $(\forall l \geq 0, t \leq m, p^{t+1}_l = p^{t}_l)$ and $(\forall l \geq m + 1, t \leq n, p^{t+1}_l = p^{t}_l)$, where $t$ is the first position at which $p^t$ differs from $p^1$. We can rewrite equation (4) to obtain an equivalent one by introducing a new function variable $a_i^{t+1}$ such that $a_i^{t+1}(p^{t+1}) = a_i^t(p^t)$, replacing $a_i(p_0^1 \ldots, p_n^1)$ by $a_i^{t+1}(p_0^1 \ldots p_n^1)$ for $t < m$. For this, we expand the domain $L_n$ by adding the integral points in the range $[p^{t+1}, p^t]$ for $0 \leq q \leq r$. It is easy to see that $p^{t+1}$ differs from $p^0$ at a maximum of $(r - q)$ points. Hence, $p^{r+1} = p^0$. By making $r$ successive iterations, we transform equation (3) into the following equivalent system of recurrence equations.

\[
a_k(p^0) = f_k(a_i(\delta_{ik}(p)), \ldots, a_s(\delta_{ik}(p)))
\]

(5)

\[
a_i^{t+1}(p^{t+1}) = a_i^t(p^t)
\]

(6)

\[
a_i^t(p^1)
\]

(7)

Repeating the same procedure over all transformations $h_{ik}$, $1 \leq i, k < s$ in equation (3), we obtain a system of directed recurrence equations. This is because, at each stage the dependency vector will be either a constant vector or a vector with only one non-zero component. Therefore, the transformation will be either non-singular or will have one dimensional kernel. □

**Proposition 1** Every normal linear recurrence equation can be mapped onto a systolic architecture.

Proof: The proof follows from Lemma (1) of [6] where we gave a method to map any DRE onto a systolic architecture. □

Henceforth we refer to normal linear recurrence equations by LRE.

### 4 Systolic Programs from LREs

The traditional approach for deriving systolic programs from LREs is to derive the timing function as discussed in the earlier section. An alternate way is to use a generalization of free schedule (cf. Karp et al. [4]) proposed in [6] for a class of LREs called DREs. We could first convert LREs into a system of equivalent DREs and apply the above method. However, this would in general introduce extraneous computation points solely for the purpose of re-routing, thereby increasing the domain size. Instead, we would like to allow existing computation points to make copies of the data, wherever necessary, for re-routing along distinct channels. This in turn would let the implementation exploit the inherent parallelism in the problem to a greater extent and would not require the introduction of too many additional computation points. In the following, we describe a procedure called CUBIZATION which implements the above strategy. The method consists of the following steps:

1. Obtaining the dependency graph of LRE,
2. Transforming the dependency graph using the new procedure called CUBIZATION which derives the timing function and
3. Embedding the dependency graph onto a systolic architecture.

Steps 1 and 3 are almost the same as in our earlier method [6]. The crux of the new method lies in the use of the second step, the CUBIZATION procedure. Before we describe it, we briefly outline step 1 and the associated definitions.

#### 4.1 Dependency graphs from LREs

The dependency graph (DG) abstracts the dependencies amongst the various functions in the LRE. The DG is formally defined below.
Definition 5 (Dependency graph) The dependency graph for a LRE is a directed labeled graph \( G(V, E, \mathcal{L}) \), as defined below:

1. Let \( V = \{(a_k, p)\} \subseteq \mathbb{L}_n \) be the set of vertices,
2. \( E \) be the set of edges, where
   \[ E = \{e_{ki}^p, e_{hki}^p\} \text{ is the edge joining } (a_i, \delta_i (p)) \text{ and } (a_k, p) \forall i,k = 1 \ldots m, p \subseteq \mathbb{L}_n \text{ and} \]
3. \( \mathcal{L} \) be a map from \( E \) to \( V \) defined by \( \mathcal{L} : e_{ki}^p \rightarrow (a_i, \delta_i(p)) \), giving labels to each edge in \( E \).

Proposition 2 The DG represents the computation induced by the linear recurrence equation.

Proof: For every variable in the LRE, there exists a vertex in the dependency graph and for every dependency between two variables in DRE, there is a directed edge between the corresponding vertices in the DG. □.

4.2 Deriving a Timing Function

For deriving the timing function, we should arrive at a deterministic mapping which yields the operations that can be evaluated at a given time. We arrive at such a function by identifying operations that could be computed concurrently without any interference with one another. As such, two children of a given node with the identical edge label will not be computed at the same time; Such a condition guarantees that one does not require global memory. On the other hand, we also have to ensure that parallelism is not precluded. For this, we constrain the ordering to only those children that belong to the same child-set (cf. Definition (7)).

Definition 6 (Direct dependence) The computation of \((a_k, p)\) is said to depend directly on \((a_i, q)\) (denoted by \((a_i, q) \rightarrow (a_k, p)\)) if there is an edge in \(G(V, E, \mathcal{L})\) directed from \((a_i, q)\) to \((a_k, p)\). We also define \(child((a_i, q)) = \{(a_k, p)\} \text{ if } (a_k, p) \text{ is a child of } (a_i, q) \text{ on an edge labeled } \mathcal{L} \).

Notation:
1. \( \preceq_{lat} \) is the lexicographic ordering defined on \( \mathbb{Z}^n \).
2. \( childbp((a_i, p), (a_j, q)) \) iff \((a_j, q) \in child((a_i, p))\).
3. \( level((a_i, p)) \) is the maximal depth of \((a_i, p)\) in the DG.

Definition 7 (Child-set) The child-set of \((a_i, p)\) under transformation \( \delta_{ij} \), denoted by \(child_{ij}(a_i, p)\), is the set of all \((a_j, q)\)'s, \( p = A_iq + b_i \), that depend directly on \((a_i, p)\).

i.e. \( child_{ij}(a_i, p) = \{(a_j, q) \mid childbp((a_i, p), (a_j, q)) \equiv true \land p = A_iq + b_i \} \) where \( b = (a_i, p) \). We also define \( childlevel(l) = \{child_{ij}(a_i, p) \mid level(a_i, p) = l\} \).

The timing function is now obtained by first transforming the DG into another graph called the Modified Dependency Graph (MDG) and then deriving the timing function for the MDG.

Algorithm to Transform DG into MDG

The CUBIZATION procedure has been designed to successfully operate on groups of child nodes of any given node in the DG. The objective is to ensure that data can be routed to all dependent nodes while ensuring that a bound can be maintained on the number of outgoing channels independent of the problem size. Initially, child nodes are grouped based on the transformations that define the dependency. Each of these classes are then recursively refined based on radix sorting on the node index (if internal dependencies already exist). At every level, base elements for each subclass are identified and ordered by applying a set of rewrite rules. Links are now introduced to forward data along the paths identified using the rewrite rules. It is easy to see that the number of output channels required is bounded from above by the dimension of the problem domain and not its size. Details of the CUBIZATION procedure are given below. Before we proceed further, we describe the preprocessing steps needed for the application of the CUBIZATION procedure.

4.2.1 Steps of the Transformation

I. Preprocessing Steps

1. All children of a given node are partitioned into classes of child-sets (cf. Definition (7)).
2. The edges leading into classes of child-sets are relabeled such that all edges leading to child nodes in the same partition have identical labels, while those leading to child nodes in different partitions have distinct labels.

II. The CUBIZATION procedure

Given an LRE, we map the dependency graph of the given equation using the CUBIZATION procedure into a MDG. The CUBIZATION procedure is given below:

```
procedure CUBIZATION(DG)
begin
  for \( h = 0 \) to \( \max Level(DG) \) do
    for each \( child_{ij}(a_i, p) \) in \( child level(h) \) do
      begin
        \( l = 0 \);
        node = SPLIT \{child_{ij}(a_i, p), \} ;
        RULE-4(DG) ;
      end
procedure CUBIZATION end
```

Procedure CUBIZATION operates on nodes of the DG and their child-sets. It calls two procedures- SPLIT to redirect
the data along distinct chains and CONNECT to connect the node to source these chains. The depth of the DG is re-evaluated at each iteration of the for loop in the procedure CUBIZATION. The for loop terminates when all parent nodes in the DG have been processed. The procedure called RULE-4 applies rewrite rule 4 [6] on the DG ensuring that a node appears at the same level in all the chains that it appears.

```plaintext
procedure SPLIT(i, cluster, [aj,p])
begin
  universe=PARTITION(cluster,i,[aj,p]);
  i-set = ∅;
  foreach partition in universe do
    begin
      if (size(partition) > 1) then
        i-set = SPLIT[i+1,partition,[aj,p]] ∪ i-set;
      else i-set = partition ∪ i-set;
      return (REWRITE(i-set,[aj,p],+i));
    end
  end
end
```
Procedure SPLIT which calls procedure PARTITION, divides the child-set into clusters based on the $j^{th}$ index. This division goes on recursively until the partition contains only one element. The variable $i$-set at recursion level $n$ contains one element per partition at the same recursion level. Using procedure REWRITE the elements of $i$-set are ordered. Procedure SPLIT returns the smallest child after application of the REWRITE procedure.

```plaintext
procedure PARTITION(cluster,i,[aj,p])
begin
  partition the cluster based on the $i^{th}$ index of p such that all elements of cluster that have the same $i^{th}$ index are in the same partition.
end
```
Procedure PARTITION divides the child-set into clusters based on the $j^{th}$ index and returns the set of clusters.

```plaintext
procedure REWRITE(set,[aj,p],l)
begin
  apply rewrite rules 1–3 [6], until it cannot be applied further to the elements of the set set, with [aj,p] as parent and label all the edges by[aj,p].wi;
  return i:childl[aj,p]; /* b = [aj,p].w */
end
```
The procedure REWRITE orders these partitions (using rewrite rules) and connects them with an edge label $<$node name>.$w, where $l$ is updated with each recursive call of procedure SPLIT (hence, all chains will have distinct labels).

```plaintext
procedure CONNECT{[ai,q],[aj,p])
begin
  if (A_j[p−q]! = 0) then RELABEL{[ai,q],[aj,p])
  else RELABEL–CHAIN{[ai,q],[aj,p])
end
```
Procedure CONNECT is used to check whether the node $(a_i,q)$ has to feed the data directly to each chain originating from it (i.e. when the point $q$ belongs to the null space of the transformation $A_j$) or to send it to the smallest child to be forwarded to the chains.

```plaintext
procedure RELABEL–CHAIN{[ai,q],[aj,p])
begin
  all edges from $[ai,q]$ labeled $[aj,p].w_i$ (except one, so that the smallest child does not get disconnected from the parent) are resourced from $(aj,p)$;
end
```
Procedure RELABEL–CHAIN routes the data directly to each chain and relabels the chains.

```plaintext
procedure RELABEL{[ai,q],[aj,p])
begin
  label the edge from $[aj,p]$ to $[ai,q]$ by $[aj,p].w_0$;
end
```
Procedure RELABEL arranges the data to the smallest child to be rerouted along distinct chains.

**Proposition 3** The transformations defined by rule 1 to rule 4 are terminating.[6]

**Proposition 4** Procedure CUBIZATION terminates.

*Proof:*

1. maxLevel is bounded.

At each level of MDG there will be at least one real node, as can be seen obviously. Hence the level of MDG is bounded by the number of nodes in DG.

2. The recursive procedure SPLIT terminates.

   Procedure PARTITION, partitions a cluster based on the $j^{th}$ index of $p$. Since the elements of a cluster are finite and $p \in L_n$, the procedure PARTITION terminates. The procedure REWRITE orders $i$-set using rewrite rules. As in Proposition (3), at each level the application of rewrite rules terminates. The maximum depth of recursion is bounded by $n$, the dimension of $L_n$. Hence procedure SPLIT terminates.

3. The procedure CONNECT terminates.

   The operation performed by procedure RELABEL–CHAIN is bounded by the out-degree of the smallest child which is bounded by $n$. RELABEL terminates as there is only one operation to be performed. It follows that CONNECT terminates.

4. The procedure RULE-4 terminates, as each application of rule 4 introduces one node and the level difference reduces by one.

(1)-(4) imply that procedure CUBIZATION terminates. □

**Proposition 5** The transformational steps preserve the dependency relation.[6]
Definition 8 (Dependence) \((a_k, p)\) is said to depend on \((a_i, q)\), denoted by \((a_i, q) \xrightarrow{\text{a}} (a_k, p)\), iff there exists a path from \((a_i, q)\) to \((a_k, p)\) labeled \((a_i, q)\) in MDG. We also define \(\text{child}^k_i(a_i, q) = \{(a_k, p)|(a_k, p)\text{ is a child of } (a_i, q) \text{ on a path labeled } b\}\).

Proposition 6 The CUBIZATION procedure preserves the dependency relation.

Proof: The CUBIZATION procedure involves the procedure SPLIT, CONNECT and RULE-4. The procedures which change the ordering in DG are procedure REWRITE, which is called by procedure SPLIT and procedure RULE-4. The procedure REWRITE defines ordering among the elements of child-set using rewrite rules 1-3 [6]. The procedure RULE-4, using rewrite rule 4 [6], introduces dummy nodes. From Proposition (5), the application of the rewrite rules 1-4 preserves the dependency relation. Hence, the lemma follows. □

Proposition 7 The MDG obtained from application of CUBIZATION procedure on DG, is a set of distinct total order chains.

Proof: The proof follows from the following observations:

1. The DG consists of a set of total order chains from Definition (5) and

2. The preprocessing steps of transformation and the application of the CUBIZATION procedure guarantee that all the children of a given node that have the same label get partitioned into child-sets, where each child-set has a distinct label (cf. Preprocessing Steps 1-2).

Next, the procedure REWRITE orders the partitions (using rewrite rules) and connects them with an edge label <nodename>.w_i, where \(l\) is updated with each recursive call of procedure SPLIT. Hence, all chains will have distinct labels. □

Definition 9 The order \(\preceq\) on MDG is defined by:

\((a_i, p) \preceq (a_j, q)\) \iff there exists a label \(\ b \in \text{MDG}\) \exists \text{child}^k_i(p((a_i, p),(a_j, q))).

Definition 10 (Chain) A chain is a maximal directed path all of whose edges have the same label. A chain is identified by the edge label.

Proposition 8 The MDG is computationally equivalent to the DG.

Proof: For every node in the DG, there exists at least one node in the MDG, because the transformation steps never delete a node. For every directed edge in the DG, there exists an equivalent directed edge or path in the MDG. This follows from the definition of the rewrite rules 1-3 which simply re-route data between two nodes and rule 4 which at most changes an edge into a path. Rule 4 ensures that a node appears at the same depth in all the chains of the MDG that contain the node and hence all the arguments needed for a computation in any given node are available at the time of its computation. The above facts combined with Proposition (6) imply the result. □

Proposition 9 \((a_k, q)\) can be computed at \(t = level(a_k, q)\).

Proof: The proof requires us to show that all arguments for a computation have been computed before hand and are available at the node by this time.

1. We observe that the arguments for the computation have already been evaluated. That is, \(\forall i, k, 1 \leq i, k \leq m\), if \((a_i, p) \stackrel{4}{\mapsto} (a_k, q)\) then \(level(a_i, p) < level(a_k, q)\), from Definition (9) and definition of level.

2. We observe that the values of the arguments computed have enough time to reach the node at which \((a_k, q)\) is computed. There exists a chain from \((a_i, p)\) to \((a_k, q)\) by the definition of MDG and the length of this path = \(level(a_k, q) - level(a_i, p)\), from rule 4 and definition of level.

3. Finally, we observe that \((a_k, q)\) can be computed at \(t = level(a_k, q)\), from induction on \(t\) and the base case that \((a_k, p_i) = v^i_k, l \in 1 \ldots l_k, l_k \in N\) and \(v^i_k\) is an initial value in \(R\) given for boundary point \(v^i_k \in L_n\). □

Lemma 2 The out-degree of every node in MDG is bounded above by a constant independent of domain size.

Proof: Every node will, at worst, be part of \(s\) (cf. equation (I)) different transformations \(\delta_i\) (or \(A_{i_j}\)) and the CUBIZATION procedure will limit the out-degree of a node for each transformation to at most \(n\), thereby the out-degree of a node is bounded above by \(n * s\). □

Lemma 3 The CUBIZATION procedure is scalable with respect to domain size.

Proof: The CUBIZATION procedure recursively divides the child-set of \((a_i, q)\) into clusters, defines an ordering among these clusters and connects them by labeled edges (using procedures SPLIT, PARTITION, REWRITE and CONNECT). If the domain size is increased, then as the ordering defined by the CUBIZATION procedure is based on the recurrence equation and the index of the points \(p, q \in L_n\), the already existing ordering will not be disturbed. □

4.2.2 Illustrative Example
Example 1 (from Quinton [8]): The following system of LREs (in normal form), computes the Gauss-Jordan diagonalization:

\[
\begin{align*}
A(i,j,k) &= a(i,j) \quad 1 \leq i \leq N, 1 \leq j \leq N, k = 0 \\
A(i,j,k) &= b(i) \quad 1 \leq i \leq N, j = N + 1, k = 0 \\
A(i,j,k) &= A(\delta_3(i,j,k)) + A(\delta_2(i,j,k)) \\
& \quad 1 \leq k \leq N, k \leq j \leq N + 1, i = k \\
X(i,j,k) &= A(\delta_3(i,j,k)) + A(\delta_2(i,j,k)) \\
& \quad 1 \leq k \leq N, k \leq j \leq N + 1, 1 \leq i < k \\
A(i,j,k) &= A(\delta_3(i,j,k)) - X(i,j,k) \\
& \quad 1 \leq k \leq N, k \leq j \leq N + 1, k < i \leq N
\end{align*}
\]

where \( \delta_3(i,j,k) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \)

The CUBIZATION procedure is applied on this example giving an MDG which embeds onto a BSA (Figure 1).

5 Mapping Onto Specific Architectures

Having obtained the computation dependencies, our next task is to map this dependency abstraction onto specific target architectures. As a first step towards mapping the computation points obtained onto a specific target architecture, we use a very general intermediate host architecture called the Basic Systolic Architecture (BSA). The BSA consists of simple and primitive processor nodes mapped onto a subset of a planar integer lattice with each node having nearest neighbor connectivity. The use of the BSA affords us a degree of architecture-independence while generating the solution. To obtain a BSA from an MDG, we have to embed all the chains of the MDG in \( Z^k \) for some \( k \). In the embedded structure, all the chains of the MDG can be identified by traversing the paths between lattice points. The process requires identifying the sub-sequences of chains that can be overlaid on other chains. The process of embedding[6] an MDG in \( Z^k \) is based on classifying the chains into parallel and coplanar chains. After classification, we embed all the chains in the MDG in \( R^k \) for some \( k \) and then apply procedure folding to get an embedding in \( Z^k \). This would in turn require that timing functions are modified to account for multiple data transmission over the same channel. The procedure staggering[6] takes care of details.

We then apply a set of semantics preserving transformations (translations and projections) to transform the BSA into the specific architecture of interest. It should be noted that these transformations may involve modification of the timing function obtained earlier.

Figure 1 shows the BSA for Gauss-Jordan diagonalization.

6 Related work and Conclusion

In this section, we compare our method described in this paper with the other approaches in the literature.

Quinton and Dongen [8] address the problem of obtaining systolic architectures from linear recurrence equations. To map linear recurrence equations onto systolic architectures, they transform the linear recurrence equation into a uniform one (uniformization), since uniform recurrence equations can be mapped onto systolic arrays [7]. The problem with such a method lies in the choice of uniformization vectors to obtain uniform system of equations having an affine timing function. The choice of uniformization vectors depends on several factors and further, to find an affine timing function one needs to obtain an integral solution to a system of linear programming problems. Our method on the other hand does not depend on finding an affine timing function. Therefore, the choice of uniformization vector does not arise in our approach. Moreover, the problem of multi-step algorithms can be handled in our approach.

Rajopadhye [9, 10], presents a technique for synthesizing systolic arrays from linear recurrence equations. He characterizes a class of transformations called data pipelining, which does not address multi-step algorithms and further, it depends on finding an affine timing function. The choice of a bad timing function could result in broadcasting, leading to poor systolic architectures. Our approach, not only handles data pipelining but also handles a system of linear recurrence equations.

Fort and Moldovan [1] discuss how or whether data broadcasts in an array processor with a given interconnection structure can be either eliminated or reduced by choosing an adequate linear schedule. Their constraints being strict, do not deal with broadcasts in their generality. Here again, the method requires the extraction of an explicit linear timing function.

Guerra and Melhem [2] have presented a method to derive systolic arrays given a specification in a canonical form. Their initial specification, though more general than UREs does not fully include LREs.

To sum up, our method is general enough to derive systolic architectures from a system of linear recurrence equations. As mentioned already, one of the distinctive features of our method is that it can map LREs onto given architectures rather than only arrive at new systolic architectures for a given LRE. This is particularly useful when we have programmable systolic architectures. Further, our method does not require an explicit timing function, is scalable with respect to domain size and can handle multi-step algorithms. We have used this method to map a few typical recurrence equations like matrix multiplication, symmetrizing Hessenberg matrix and Gauss-Jordan elimination onto standard architectures successfully.
For details we refer the reader to [5] or the full version of the paper available at the URL ftp://ftp.tcs.tifr.res.in/pub/papers/landan/ipps96-full.ps.gz.

Post Script: In a recent talk [11], Anant Agarwal described a methodology wherein algorithms are decomposed into equivalent logic circuits (represented as netlists) and then embedded onto a matrix of FPGAs to speed up computation. It is of interest to note that our method for routing of signals as well as sharing of communication links between nodes of the BSA can be applied to this problem domain in an effective way.

References


Figure 1: Basic systolic architecture for Gauss-Jordan diagonalization.