Algorithm-Specific Parallel Processing with Linear Processing Arrays

JOSE A. B. FORTES

School of Electrical Engineering
Purdue University
West Lafayette, Indiana

BENJAMIN W. WAH

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois

WEIJIA SHANG

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, Louisiana

KUMAR N. GANAPATHY

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois

1. Introduction

Many applications of digital signal processing, scientific computing, digital communications, and control are characterized by repeated execution of a small number of computationally intensive operations. In order to meet performance requirements it is often necessary to dedicate hardware with parallel processing capabilities to these specialized operations. Processor arrays, due to their structural regularity and consequent suitability for VLSI implementation, are frequently used for this purpose. Regardless of whether their structures are fixed or designed to match algorithm characteristics, it is important to understand how to map these algorithms into processor arrays. This article discusses systematic ways of deriving such mappings. The techniques are illustrated by examples involving linear arrays of processors (one-dimensional processor arrays); however, unless otherwise stated, the results can be extended to arrays of arbitrary dimension.

Several linear arrays have been implemented for specific applications as well as for "wide-purpose" computing (Valero *et al.*, 1991; Fortes *et al.*, 1992). They are easier to build and program than arrays of higher dimensions. In particular, the connections among neighboring processors can be made very fast and, therefore, provide large communication bandwidths. For example, physical links in Warp (Annaratone *et al.*, 1987; Menzilcioglu *et al.*, 1989; Baxter *et al.*, 1990) can serve several virtual channels capable of communication among neighboring cells. This is also the case for algorithm-specific linear arrays but, for design simplicity, each link may be used to transfer only a data item instead of being multiplexed among several of them. In this paper, we present two methods of systematically mapping recurrent computations on such linearly connected processor arrays. The first method guarantees that no more than one computation is assigned to execute in any processor at any given time but assumes enough data channels between processors for the necessary data communication. The second method considers, in addition to computational conflicts, the possibility of communication conflicts and guarantees that individual links are not required to pass more than one data item at a time.

The techniques discussed here apply to algorithms described as recurrences, either by mathematical expressions or by high-level language programs. Section 1.2 provides a precise characterization of the class of algorithms for which our results are strictly valid. However, more general classes of algorithms and programs can also be mapped using similar techniques either in a piecewise manner or as heuristics to guide the search for good designs.

Once algorithms are characterized as sets of computations ordered by data dependences, the problem of mapping algorithms to processor arrays becomes equivalent to that of finding a function that assigns a processor and an instant of time to each computation of that algorithm. This function must have certain properties that guarantee computational correctness and efficient usage of processing resources. The techniques described in this paper include techniques and methods of mapping parameterized representations of the algorithms to linearly connected architectures of interest. Sections 2, 3, and 4 discuss the *dependence-based method* (DM) and techniques of selecting linear mappings. Sections 5 and 6 present an alternative approach, called the *generalized parameter-based method* (GPM), and show how it relates to the dependence-based method. Optimization procedures used with both approaches are described along with examples. Sections 2–4 are partially based on the work of Shang and Fortes (1992), and Sections 5 and 6, on that of Ganapathy and Wah (1992a,b).

1.1 General Notation

Arrows are used to denote vectors, while the *transpose* of a vector $\vec{v}$ or of a matrix M are denoted $\vec{v}^T$ and M^T, respectively. The notation $\vec{v} \geq \vec{u}$ means every component of $\vec{v}$ is greater than or equal to the corresponding component of $\vec{u}$. The vector $\vec{0}$ denotes a row or column vector whose entries are all zeros. The dimensions of vector $\vec{0}$ and whether it denotes row or column vectors are implied by the context in which the vector is used. The rank and determinant of a matrix A are denoted *rank*(A) and *det*(A), respectively. The set of integers is denoted by $\mathbf{Z}$, the empty set by $\varnothing$, the cardinality of a set C by $|C|$, and the absolute values of a scalar α by $|\alpha|$.

1.2 Algorithm Model

Uniform dependence algorithms (or uniform recurrence equations—UREs) are formally defined as follows.

Definition 1.1 (Uniform Dependence Algorithm). A *uniform dependence algorithm* is an algorithm that can be described by an equation of the form

$$v(\vec{I}) = g_{\vec{I}}(v(\vec{I} - \vec{d}_1), v(\vec{I} - \vec{d}_2), \ldots, v(\vec{I} - \vec{d}_r)), \tag{1.1}$$

where

1. $\vec{I} = [i_1, \ldots, i_n]^T \in J \subset \mathbf{Z}^n$ is an index point (a column vector); J is the *index set* of the algorithm; and n is the number of components of $\vec{I}$;

2. $g_{\vec{I}}$ is a computation indexed by $\vec{I}$, i.e., a single-valued function computed "at index point $\vec{I}$" in a *single unit of time*;
3. $v(\vec{I})$ is the value computed "at $\vec{I}$," i.e., the result of computing the right-hand side of (1.1); output variables correspond to values at particular index points $\vec{I}$; if $\vec{I}$ is not in J, then $v(\vec{I})$ is an input variable;
4. $\vec{d}_j$, $j = 1, \ldots, r$ are constant vectors (i.e., independent of $\vec{I} \in J$) called *dependences*; the matrix $D = [\vec{d}_1, \ldots, \vec{d}_r]$ is called a *dependence matrix*. ■

A well-known simple example of an URE is

$$c(i_1, i_2, i_3) = c(i_1, i_2, i_3 - 1) + a_{i_1, i_3} b_{i_3, i_2} \quad 1 \le i_1, i_2, i_3 \le N, \qquad (1.2)$$

which describes the computation of an $(N \times N)$ matrix C as the product of two $(N \times N)$ matrices A and B. This algorithm will be used as a running example throughout the description of the dependence-based method. The index set consists of all the integer points within a cube with sides of length N. At each point $\vec{I} = [i_1, i_2, i_3]^T$ a distinct one-variable function specified by the operator "$+a_{i_1, i_3} b_{i_3, i_2}$" is computed (other representations where a_{i_1, i_3} and b_{i_3, i_2} are treated as variables and the operator is simply "+" are discussed below). There is a single dependence $[0, 0, 1]^T$, and the product matrix corresponds to the values of $c(i_1, i_2, N)$.

Uniform dependence algorithms can be found in many scientific computations, digital signal processing applications, and other fields. However, a much larger class of algorithms, called *affine dependence algorithms* (or *affine recurrence equations—AREs*), can also benefit from the techniques proposed for UREs. For AREs, (1.1) is replaced by:

$$v(h(\vec{I})) = g_{\vec{I}}(v(f_1(\vec{I})), v(f_2(\vec{I})), \ldots, v(f_r(\vec{I}))), \qquad (1.3)$$

where the indices h and f_j, $j = 1, \ldots, r$, are affine functions of $\vec{I}$, i.e., functions of the form $F\vec{I} + \vec{c}$, where F is a matrix with n columns and $\vec{c}$ is a constant vector with as many elements as the number of rows in F. There exist techniques to transform AREs to UREs (that is, to *uniformize* AREs) but they are outside the scope of this article. The basic idea there is to select a few integral basis vectors such that all affine dependence vectors of the ARE can be expressed as nonnegative integer linear combinations of the basis vectors. These vectors correspond to uniform dependences following the uniformization. For example, in the URE shown in (1.2) for matrix multiplication, a_{i_1, i_3} and b_{i_3, i_2} are inputs that are used in several computations (e.g., $a_{1,1}$ is used in computations to generate $c(1, i_2, 1)$ for all values of i_2). This can be shown explicitly as follows:

$$
\begin{aligned}
a(i_1, i_2, i_3) &= a(i_1, 0, i_3) && 1 \le i_1, i_2, i_3 \le N \\
b(i_1, i_2, i_3) &= b(0, i_2, i_3) && 1 \le i_1, i_2, i_3 \le N \\
c(i_1, i_2, i_3) &= c(i_1, i_2, i_3 - 1) + a(i_1, i_2, i_3)b(i_1, i_2, i_3) && 1 \le i_1, i_2, i_3 \le N
\end{aligned}
\tag{1.4}
$$

where $a(i_1, 0, i_3) = a_{i_1, i_3}$ and $b(0, i_2, i_3) = b_{i_3, i_2}$. The dependences for the first two statements are affine, i.e., they are $[0, i_2, 0]^T$ and $[i_1, 0, 0]^T$, respectively. Reusing or "pipelining" of these data among different computations can be done as follows (yielding a decomposition of the affine dependences in terms of the basis vectors $[0, 1, 0]^T$ and $[1, 0, 0]^T$ that are also uniform dependences resulting from the uniformization).

$$
\begin{aligned}
a(i_1, i_2, i_3) &= a(i_1, i_2 - 1, i_3) && 1 \le i_1, i_2, i_3 \le N \\
b(i_1, i_2, i_3) &= b(i_1 - 1, i_2, i_3) && 1 \le i_1, i_2, i_3 \le N \\
c(i_1, i_2, i_3) &= c(i_1, i_2, i_3 - 1) + a(i_1, i_2, i_3)b(i_1, i_2, i_3) && 1 \le i_1, i_2, i_3 \le N
\end{aligned}
\tag{1.5}
$$

This simple example illustrates also another advantage of uniformization in which we can eliminate broadcasts of data to many processing elements. In the above uniformizing algorithm, distinct variables (which may have identical values) are used to compute distinct $c(i_1, i_2, i_3)$. Procedures for uniformization and broadcast removal share many similarities and are discussed in Chen and Shang (1992), Quinton and Van Dongen (1989), Tzen and Ni (1992), Wong and Delosme (1992), and Yaacoby and Cappello (1988).

The matrix multiplication example also illustrates the fact that in a single algorithm there may be more than one recurrence equation of the form of (1.1). For the purpose of this article, we view any system of recurrence equations as a single recurrence that computes a tuple whose elements are variables on the left-hand sides of the individual recurrences of the system. For example, the transformed UREs in (1.5) can be described by a single recurrence:

$$
\begin{aligned}
&[a, b, c](i_1, i_2, i_3) \\
&\quad = g([a, b, c](i_1, i_2 - 1, i_3), [a, b, c](i_1 - 1, i_2, i_3), [a, b, c](i_1, i_2, i_3 - 1)) \\
&\quad = [a(i_1, i_2 - 1, i_3), b(i_1 - 1, i_2, i_3), c(i_1, i_2, i_3 - 1) \\
&\qquad + a(i_1, i_2, i_3)b(i_1, i_2, i_3)].
\end{aligned}
\tag{1.6}
$$

Note that we have used an ordered set notation to represent all the variables used in a system of recurrences (instead of representing each variable separately).

1.3 Relation to Nested-Loop Programs

Affine dependence algorithms are common in image processing, digital signal processing, and other scientific applications where regular computation-intensive operations are required (Fortes and Wah, 1987; Tucker and Robertson, 1988; Almasi and Gottlieb, 1989). In practice many of the algorithms that are executed by processor arrays are described in a procedural high-level language such as Fortran. Nested loops are often the most time-consuming kernels of these programs and are, therefore, targets of hardware accelerators based on processor arrays. It turns out that a large number of Fortran-like nested loops can be modeled as affine recurrences. For instance, it is relatively easy to relate the following generic nested loop to a corresponding system of affine recurrences.

$$
\begin{array}{l}
\text{DO } i_1 = l_1 \text{ TO } N_1 \\
\quad \text{DO } i_2 = l_2 \text{ TO } N_2 \\
\quad \cdots \\
\quad\quad \text{DO } i_n = l_n \text{ TO } N_n \\
\quad\quad\quad S_1(\vec{I}) \\
\quad\quad\quad S_2(\vec{I}) \\
\quad\quad\quad \cdots \\
\quad\quad\quad S_p(\vec{I}) \\
\quad\quad \text{END} \\
\quad \cdots \\
\quad \text{END} \\
\text{END}
\end{array}
\tag{1.7}
$$

where $S_j(\vec{I})$ contains an assignment of the form of (1.3), i.e.,

$$v_j(h_j(\vec{I})) = g_j(v_1(f_{j,1}(\vec{I})), \ldots, v_p(f_{j,p}(\vec{I}))). \tag{1.8}$$

It is possible that variable v_j, $1 \le j \le p$, appears more than once with different indexing functions such as $v_1(i, j) = v_1(i + j, i - j) + v_1(2i + j, 3i)$. Each appearance of a variable on the right-hand side may cause a dependence (Banerjee, 1988; Xing and Shang, 1993). If all the loop bounds l_j and N_j, $j = 1, \ldots, n$, are linear functions of the index variables $i_1, \ldots, i_{j-1}$, the set of all the iteration vectors $\vec{I}$ of the loop can be described by a convex polyhedron. Affine recurrences can be used to model the program in (1.7) when (1) the indexing functions $h_j(\vec{I})$ and $f_{j,l}(\vec{I})$, $j, l = 1, \ldots, p$, are affine; (2) the bounds l_j and N_j, $j = 1, \ldots, n$, are affine functions of the index variables $i_1, \ldots, i_{j-1}$; and (3) branch statements are allowed as long as all branches cause the same dependences, the

computation times for different branches are the same, and branches do not go outside the loop.

For example, it is easy to see that the following nested-loop program corresponds to the pipelined version of the matrix multiplication algorithm described in (1.5).

$$
\begin{aligned}
&\text{DO } i_1 = 1 \text{ TO } N \\
&\quad \text{DO } i_2 = 1 \text{ TO } N \\
&\quad\quad \text{DO } i_3 = 1 \text{ TO } N \\
&\quad\quad\quad a(i_1, i_2, i_3) = a(i_1, i_2 - 1, i_3) \\
&\quad\quad\quad b(i_1, i_2, i_3) = b(i_1 - 1, i_2, i_3) \\
&\quad\quad\quad c(i_1, i_2, i_3) = c(i_1, i_2, i_3 - 1) + a(i_1, i_2, i_3)b(i_1, i_2, i_3) \\
&\quad\quad \text{END} \\
&\quad \text{END} \\
&\text{END}
\end{aligned}
\tag{1.9}
$$

Intuitively, datum a_{i_1, i_3} is pipelined along the i_2 axis from index point $[i_1, 1, i_3]^T$ to $[i_1, 2, i_3]^T, \ldots,$ and to $[i_1, N, i_3]^T$. Similarly, b_{i_3, i_2} is pipelined along the i_1 axis. Initially, $a(i_1, 0, i_3) = a_{i_1, i_3}$ and $b(0, i_2, i_3) = b_{i_3, i_2}$.

In short, we focus in this article on algorithms that can be modeled as uniform recurrences and affine recurrences that can be uniformized. For the purpose of this paper, only structural information of the algorithm, i.e., the index set J and dependence matrix D, is needed. However, when addressing the problem of avoiding data-link conflicts, we use information about input/output data distributions derived from the desired mapping. Note that all computations are identical in the processor array when uniform recurrences are mapped. When inputs/outputs occur at the peripheral boundary of the processor array we need only consider a very limited number of possible data distributions of inputs/outputs. A uniform dependence algorithm with dependence matrix D and index J is, therefore, described by a pair (J, D). For the matrix-multiplication algorithm in (1.5) or (1.9), we have

$$
J = \left\{ \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} : 1 \le i_1, i_2, i_3 \le N,\ i_1, i_2, i_3 \in \mathbf{Z} \right\} \quad \text{and} \quad D = \begin{array}{c} \begin{matrix} b & a & c \end{matrix} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}. \tag{1.10}
$$

The symbol on the top of each column in D indicates the variable that causes the dependence.

2. The Mapping Problem

The central problem addressed in this paper is that of mapping an n-dimensional algorithm into an m-dimensional processor array where $m \leq n - 1$. While the examples of this paper emphasize the case of linear arrays (i.e., $m = 1$), the technical discussions are valid for larger values of m, i.e., our techniques are applicable to processor arrays of arbitrary dimension. The mappings of interest are *linear* in nature and characterized by a *mapping matrix* of the following form:

$$T = \begin{bmatrix} S \\ \Pi \end{bmatrix} \in \mathbf{Z}^{(m+1)\times n} \tag{2.1}$$

such that the computation indexed by $\vec{I}$ is executed at time $\Pi\vec{I}$ by processor $S\vec{I}$. The vector Π is called the *time schedule vector*, and S, the *allocation matrix*. Valid mappings must satisfy the following conditions:

1. *Causality*—$\Pi D > \vec{0}$.
   This condition ensures the correct ordering of computations so that a computation at index $\vec{I}$ is never executed before a computation at $\vec{I} - \vec{d}_j$, $j = 1, \ldots, r$, i.e., a computation never takes place before its operands are available.
2. *Routability*—$SD = PK$.
   $P \in \mathbf{Z}^{m\times w}$ is the matrix of interconnection primitives of the processor array and $K \in \mathbf{Z}^{w\times m}$ is defined in such a way that

$$\sum_{i=1}^{w} k_{i,j} \leq \Pi\vec{d}_j, \qquad j = 1, \ldots, r. \tag{2.2}$$

   The matrix P describes the array connectivity. For example, an array in which each processor is connected to the four nearest east, south, west, and north neighbors has

$$P = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{bmatrix}.$$

   A linear array where each processor has connections to its left and right neighbors has $P = [1, -1]$. The term $\sum_{i=1}^{w} k_{i,j}$ represents the number of times the primitives must be used to route the datum associated with the dependence $\vec{d}_j$. This sum must be less than or equal to the interval of time between generation and consumption of the datum in order for the processor array to be able to implement the time schedule vector Π.

3. *Conflict-free computations*—for all $\vec{I}_1, \vec{I}_2 \in J$, if $\vec{I}_1 \neq \vec{I}_2$, then $T\vec{I}_1 \neq T\vec{I}_2$. This condition guarantees that no processor has to execute more than one computation at any given time.
4. *Rank compatibility*—$rank(T) = m + 1$. This ensures that the algorithm is mapped into an array of m dimensions, since $rank(T) - 1$ is the dimension of the array derived from T.

Additional constraints are possible but are not considered here. They depend on implementation requirements. For instance, Lee and Kedem (1988, 1990), Ganapathy and Wah (1992a,b) (to be discussed in Sections 5 and 6), and Xue (1993) have introduced constraints that guarantee that data collisions do not occur in communication links.

The execution time that results from using mapping matrix T is given by

$$t = \max\{\Pi(\vec{I}_1 - \vec{I}_2) : \vec{I}_1, \vec{I}_2 \in J\} + 1, \tag{2.3}$$

which reduces to

$$t = 1 + \sum_{j=1}^{n} |\pi_j|(N_j - l_j) \tag{2.4}$$

when index set J is *constant bounded*; that is, J is of the form

$$J = \{[i_1, \ldots, i_n]^T : l_j \leq i_j \leq N_j,\ i_j \in \mathbf{Z},\ j = 1, \ldots, n\}, \tag{2.5}$$

where l_j and N_j correspond to the constant lower and upper bounds of the jth loop, respectively. An important property of (2.4) is that the total execution time is a monotonically increasing function of $|\pi_j|$, $j = 1, \ldots, n$ (O'Keefe and Fortes, 1986; O'Keefe *et al.*, 1991; Li and Wah, 1985). This fact implies that it is not possible to reduce the absolute value of any of the entries of the optimal time schedule vector Π without obtaining an invalid mapping.

When $rank(T) = n$, computation conflicts can be avoided by ensuring that T is nonsingular because the mapping described by T is then injective. However, when $rank(T) = m + 1 < n$, it is always possible to have $T\vec{I}_1 = T\vec{I}_2$ for two different points $\vec{I}_1$ and $\vec{I}_2$ (or equivalently $T(\vec{I}_1 - \vec{I}_2) = \vec{0}$), and if these points belong to the index set, a conflict occurs in the corresponding computations. To avoid conflicts, it is desirable to use a mapping matrix T such that for any $\vec{I} \in J$ and any nonzero integer solution $\vec{\gamma}$ of $T\vec{\gamma} = \vec{0}$, the point $\vec{I} + \vec{\gamma}$ does not belong to J. It is easy to show that we need to consider only those solutions of $T\vec{\gamma} = \vec{0}$ whose entries are relatively prime, i.e., $gcd(\gamma_1, \ldots, \gamma_n) = 1$.[1] These solutions are called *conflict vectors*. For a given conflict vector $\vec{\gamma}$ and any arbitrary index point $\vec{I}$, if $\vec{I} + \vec{\gamma} \notin J$,

[1] $gcd(a_1, \ldots, a_n)$ denotes the greatest common divisor of integers $a_1, \ldots, a_n$.

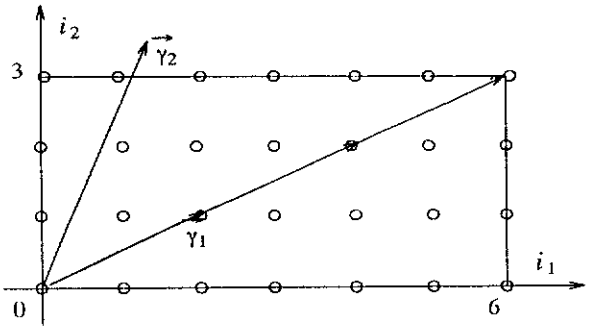

FIG. 1. Nonfeasible and feasible conflict vectors $\vec{\gamma}_1$ and $\vec{\gamma}_2$.

the vector $\vec{\gamma}$ is a *feasible conflict vector*; otherwise, it is *nonfeasible*. If all conflict vectors of a mapping matrix T are feasible, the mapping is *computation conflict free*.

This concept is illustrated in Fig. 1, which shows a two-dimensional index set $J = \{[i_1, i_2]^T : 0 \le i_1 \le 6, 0 \le i_1 \le 3, i_1, i_2 \in \mathbf{Z}\}$. For the conflict vector $\vec{\gamma}_1 = [2, 1]^T$, both $\vec{I} = \vec{0}$ and $\vec{I} + \vec{\gamma}_1 = [2, 1]^T$ belong to index set J, and computations indexed by $[0, 0]^T$, $[2, 1]^T$, $[4, 2]^T$, and $[6, 3]^T$ are mapped to the same processor and the same execution time. Therefore, there is at least one conflict. However, for conflict vector $\vec{\gamma}_2 = [2, 5]^T$, there will be no conflict at all because for any arbitrary $\vec{I} \in J$, we have $\vec{I} + \vec{\gamma}_2 \notin J$. Intuitively, if the vector $[2, 5]^T$ is drawn with one end at $[0, 0]^T$ (or at any other index point of the index set), the other end is outside the index set, and the vector $[2, 5]^T$ does not meet any integer points in the index set. Therefore, the mapping with this $\vec{\gamma}$ is conflict free.

As another example, consider a four-dimensional algorithm (J, D), where

$$J = \{\vec{I} : \vec{I} \in \mathbf{Z}^4, 0 \le i_j \le 7, j = 1, \ldots, 4\}. \tag{2.6}$$

Assuming that this algorithm is to be mapped to a one-dimensional (linear) processor array, one possible mapping matrix is

$$T = \begin{bmatrix} 1 & 1 & 8 & 1 \\ 0 & 1 & 8 & 1 \end{bmatrix}. \tag{2.7}$$

Consider the following solutions of $T\vec{\gamma} = \vec{0}$: $\vec{\gamma}_1 = [0, 8, -1, 0]^T$, $\vec{\gamma}_2 = [0, 0, 1, -8]^T$, and $\vec{\gamma}_3 = [0, 1, 0, -1]^T$. Clearly, $T\vec{\gamma}_1 = T\vec{\gamma}_2 = T\vec{\gamma}_3 = \vec{0}$, and the greatest common divisor of their entries is unity. So $\vec{\gamma}_1$, $\vec{\gamma}_2$, and $\vec{\gamma}_3$ are the conflict vectors of the mapping matrix T. However, the vector $[0, 2, 0, -2]^T$ is also a solution of the equation $T\vec{\gamma} = \vec{0}$, but is not a conflict

vector of the mapping matrix T because the greatest common divisor of its entries is not unity. The conflict vectors $\vec{\gamma}_1$ and $\vec{\gamma}_2$ are feasible because for any arbitrary index point $\vec{I} \in J$, we have $\vec{I} + \vec{\gamma}_j \notin J$, $j = 1, 2$. The conflict vector $\vec{\gamma}_3$ is not feasible because for index point $\vec{I} = [0, 0, 0, 1]^T \in J$, we have $\vec{I} + \vec{\gamma}_3 = [0, 1, 0, 0]^T \in J$. Therefore, T has computational conflicts.

It is not practical to check that every conflict vector $\vec{\gamma}$ of a mapping matrix T is feasible (this involves checking that $\vec{I} + \vec{\gamma}$ is not in J for every computation $\vec{I}$ of the algorithm). Therefore, conditions under which T is conflict free are discussed in the next section. In Section 4 the problem of finding optimal conflict-free mappings given an allocation matrix is addressed.

3. Computation-Conflict-Free Mappings

For constant-bounded algorithms of size $N_j - l_j$ (see (2.5)), it is simple to show that a mapping matrix T is conflict free if and only if every conflict vector $\vec{\gamma}$ has at least one entry γ_j such that $\gamma_j > N_j - l_j$ (Shang and Fortes, 1992). However, it is desirable to express these conditions in terms of the time mapping vector Π so that it is possible to use the conditions in order to select the optimal schedule. To understand this concept, we first discuss the easiest case of $m = n - 2$.

Let $\Pi \in \mathbf{Z}^{1\times n}$, $S \in \mathbf{Z}^{(n-2)\times n}$, and $rank(S) = n - 2$. Consider the following equation

$$T\vec{\gamma} = \vec{0} \quad \text{or} \quad \begin{bmatrix} S \\ \Pi \end{bmatrix} \vec{\gamma} = \vec{0}. \tag{3.1}$$

We first assume that $rank(T) = n - 1$. Later in this section, we give conditions on Π that guarantee $rank(T) = n - 1$. Clearly, there is only one linearly independent solution of (3.1). Without loss of generality, let $T = [B, \vec{b}]$, where B contains the first $n - 1$ columns of T, $rank(B) = n - 1$, and $\vec{b}$ is the last column of T. Also, let B^* and $det(B)$ be the adjugate matrix and determinant of matrix B, respectively (Strang, 1980, p. 170). Then all the solutions of (3.1) can be expressed as

$$\vec{\gamma} = \lambda \begin{bmatrix} -B^*\vec{b} \\ det(B) \end{bmatrix} = \lambda \begin{bmatrix} f_1(\pi_1, \ldots, \pi_n) \\ f_2(\pi_1, \ldots, \pi_n) \\ \cdots \\ f_n(\pi_1, \ldots, \pi_n) \end{bmatrix}, \tag{3.2}$$

where λ is a constant.

If the first nonzero entry of a conflict vector is assumed to be positive (which does not entail any loss of generality), then for the mapping matrix $T \in \mathbf{Z}^{(n-1)\times n}$, there is only one unique conflict vector (otherwise, $-\vec{\gamma}$ would also be a conflict vector). This unique conflict vector $\vec{\gamma}$ is expressed by (3.2), where λ is such that $\vec{\gamma}$ is integral, its entries are relatively prime, and the first nonzero entry is positive. If this unique conflict vector is feasible, the corresponding mapping is conflict free. In addition, if Π is such that there exists a nonzero entry $f_j(\pi_1, \ldots, \pi_n)$, $1 \le j \le n$, $rank(T) = n - 1$ because $f_j(\pi_1, \ldots, \pi_n)$ is the determinant of the submatrix of T consisting of all but the jth columns of T (Strang, 1980).

Ideally, the functions f_j in (3.2) would be linear, and the feasibility constraints mentioned above ($|\gamma_j| > N_j - l_j$ for some j) would allow the identification of an optimal mapping by solving an integer linear programming problem. It turns out that if the allocation matrix S is known, f_j, $j = 1, \ldots, n$, are indeed linear functions of π_l, $l = 1, \ldots, n$. This is illustrated by the following example.

Example 3.1. Consider the matrix-multiplication algorithm. If the space allocation matrix S is chosen as $[1, -1, 0]$, the mapping matrix T and its conflict vector $\vec{\gamma}$ are

$$T = \begin{bmatrix} 1 & -1 & 0 \\ \pi_1 & \pi_2 & \pi_3 \end{bmatrix}, \qquad \vec{\gamma} = \lambda \begin{bmatrix} \pi_3 \\ \pi_3 \\ -(\pi_1 + \pi_2) \end{bmatrix}. \tag{3.3}$$

It is clear that $T\vec{\gamma} = \vec{0}$. If Π is chosen such that $\pi_3 \neq 0$ or $\pi_1 + \pi_2 \neq 0$, then $rank(T) = n - 1 = 2$. ■

Consider now conditions for the general case where $m \le n - 2$. In these mappings, $T \in \mathbf{Z}^{(m+1)\times n}$, $T = \begin{bmatrix} S \\ \Pi \end{bmatrix}$, $\Pi \in \mathbf{Z}^{1\times n}$, and $S \in \mathbf{Z}^{m\times n}$. Consider again (3.1). If $rank(T) = m + 1$, there are $n - (m + 1)$ linearly independent solutions of (3.1). Let $\vec{\gamma}_1, \ldots, \vec{\gamma}_{n-(m+1)}$ be the linearly independent integral solutions of (3.1) whose entries are relatively prime. All solutions $\vec{\gamma}$ of (3.1) can be represented as the following linear combinations:

$$\vec{\gamma} = \lambda_1 \vec{\gamma}_1 + \cdots + \lambda_{n-(m+1)} \vec{\gamma}_{n-(m+1)}. \tag{3.4}$$

Clearly, $\vec{\gamma}_1, \ldots, \vec{\gamma}_{n-(m+1)}$ are conflict vectors of T. In general, the mapping matrix T has more than $n - (m + 1)$ conflict vectors when $m < n - 2$ because a linear combination of these $n - (m + 1)$ conflict vectors may represent a different integral vector, whose entries are relatively prime, and, therefore, is another conflict vector of T. This new conflict vector may or may not be feasible. Thus, unlike the mapping matrix $T \in \mathbf{Z}^{(n-1)\times n}$

described earlier in this section, we cannot guarantee that all conflict vectors of T are feasible even if the $n - (m + 1)$ linearly independent solutions $\vec{\gamma}_j$, $j = 1, \ldots, n - (m + 1)$, of the equation $T\vec{\gamma} = \vec{0}$ are all feasible. This is illustrated by the following example.

Example 3.2. Consider the algorithm with the four-dimensional index set in (2.6) and mapping matrix T in (2.7). Let $\vec{\gamma}_1 = [0, 8, -1, 0]^T$ and $\vec{\gamma}_2 = [0, 0, 1, -8]^T$. Clearly, $T\vec{\gamma}_1 = T\vec{\gamma}_2 = \vec{0}$, and $\vec{\gamma}_1$ and $\vec{\gamma}_2$ are linearly independent and are feasible conflict vectors of T. Let $\vec{\gamma} = \vec{\gamma}_1/8 + \vec{\gamma}_2/8 = [0, 1, 0, -1]^T$. The vector $\vec{\gamma}$ is also a solution of the equation $T\vec{\gamma} = \vec{0}$ whose entries are relatively prime and, therefore, is a conflict vector of T. Because none of the absolute values of the entries of $\vec{\gamma}$ is greater than the corresponding dimension size $N_j - l_j = 7$, $\vec{\gamma}$ is not feasible in the sense discussed at the beginning of the present section. Therefore, as already mentioned, for a given mapping matrix $T \in \mathbf{Z}^{(m+1)\times n}$ with $m < n - 2$, there are possibly more than $n - (m + 1)$ conflict vectors, and T may not be conflict free even if there are $n - (m + 1)$ linearly independent feasible conflict vectors of T. ■

Example 3.2 brings out the difficulties involved in making all the conflict vectors of a mapping matrix T feasible. Nonfeasible conflict vectors can result from rational linear combinations of the $n - (m + 1)$ linearly independent feasible conflict vectors $\vec{\gamma}_1, \ldots, \vec{\gamma}_{n-(m+1)}$ (as is illustrated by $\vec{\gamma} = \vec{\gamma}_1/8 + \vec{\gamma}_2/8$ in Example 3.2). However, there is another way of selecting the $n - (m + 1)$ linearly independent conflict vectors of T such that the constants λ_j, $j = 1, \ldots, n - (m + 1)$, in (3.4) must be integral in order for $\vec{\gamma}$ to be integral. As we now explain, the Hermite normal form (Schrijver, 1986, p. 45) of the mapping matrix T can be used to achieve this.

For any matrix $T \in \mathbf{Z}^{(m+1)\times n}$ with $rank(T) = m + 1$, there exists a unimodular[2] matrix $U \in \mathbf{Z}^{n\times n}$ such that $TU = H = [L, \mathbf{0}]$, where $\mathbf{0}$ denotes a zero-entry matrix, and $L \in \mathbf{Z}^{(m+1)\times(m+1)}$ is a nonsingular and lower triangular matrix whose diagonal elements are positive and each of whose diagonal elements is the maximum of all the absolute values of the elements in that same row (Schrijver, 1986, p. 45). The matrix H is called the *Hermite normal form* of T. For the purpose of this paper it is enough to know that T can be transformed into a lower triangular matrix $[L, \mathbf{0}]$ by right multiplication of a unimodular matrix U. It is not required that each diagonal element of L be positive or be the maximum of all the absolute values of the elements in that same row.

[2] A matrix is *unimodular* if and only if it is integral and the absolute value of its determinant is unity.

For a given mapping matrix T, let H be the corresponding Hermite normal form and $T = HV$, where $V = U^{-1}$, $U = [\vec{u}_1, \ldots, \vec{u}_n]$, and $V = [\vec{v}_1, \ldots, \vec{v}_n]$. Equation (3.1) can be rewritten as $HV\vec{\gamma} = \vec{0}$. Let $\vec{\beta} = V\vec{\gamma} = [\beta_1, \ldots, \beta_n]^T$ and $\vec{\gamma} = U\vec{\beta}$. Then the following theorem is true (Shang and Fortes, 1992).

Theorem 3.1.

1. *$H\vec{\beta} = \vec{0}$ if and only if $\beta_1, \ldots, \beta_{m+1}$ are all zero.*
2. *The vector $\vec{\gamma}$ is integral if and only if $\vec{\beta}$ is integral.*
3. *The vector $\vec{\gamma}$ is a conflict vector of the mapping matrix T if and only if*

$$\vec{\gamma} = [\vec{u}_{m+2}, \ldots, \vec{u}_n] \begin{bmatrix} \beta_{m+2} \\ \cdots \\ \beta_n \end{bmatrix}, \tag{3.5}$$

where β_j, $j = m + 2, \ldots, n$, are arbitrary integers that are relatively prime and not all zero.

What Theorem 3.1 implies is that all the conflict vectors of the mapping matrix T can be represented by (3.5), where $\beta_{m+2}, \ldots, \beta_n$ are arbitrary integers that are relatively prime and not all zero. Notice that a nonintegral value of any one of the $\beta_{m+2}, \ldots, \beta_n$ results in a nonintegral vector $\vec{\gamma}$ according to Theorem 3.1. Hence, in this representation, we can avoid the case where a new conflict vector of T is obtained by a nonintegral linear combination of the $n - (m + 1)$ linearly independent solutions of (3.1).

Example 3.3. The Hermite normal form of the mapping matrix T in (2.7) is

$$TU = H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

where

$$U = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 0 & -8 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad V = U^{-1} = \begin{bmatrix} 1 & 1 & 8 & 1 \\ 0 & 1 & 8 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & -8 & 0 \end{bmatrix}.$$

All the conflict vectors of T are integral combinations of the third and fourth columns of the matrix U as follows:

$$\vec{y} = \begin{bmatrix} 0 & 0 \\ -8 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_3 \\ \beta_4 \end{bmatrix},$$

where β_3 and β_4 are integers that are relatively prime and are not both zero. ■

The Hermite normal form of T provides a convenient representation of all conflict vectors. The following two theorems provide necessary conditions for conflict-free computations based on the entries of the matrix U. This matrix can be computed in polynomial time (Kannan and Bachem, 1979) and when the allocation matrix S is known, it is possible to express the entries of U as functions of the time schedule Π.

Theorem 3.2. *Let $v_{i,j}$ be the entry of a matrix V at the ith row and the jth column. If the mapping matrix T is conflict free, at least one of its first $m+1$ entries of each and every column of V must be nonzero; that is, the following conditions hold:*

$$\begin{aligned} &(v_{1,1} \neq 0 \vee v_{2,1} \neq 0 \vee \cdots \vee v_{m+1,1} \neq 0) \wedge \\ &(v_{1,2} \neq 0 \vee v_{2,2} \neq 0 \vee \cdots \vee v_{m+1,2} \neq 0) \wedge \\ &\quad \cdots \\ &(v_{1,n} \neq 0 \vee v_{2,n} \neq 0 \vee \cdots \vee v_{m+1,n} \neq 0). \end{aligned} \tag{3.6}$$

Theorem 3.3. *If the mapping matrix T is feasible, $\vec{u}_{m+2}, \ldots, \vec{u}_n$ are feasible conflict vectors.*

It is also possible to derive sufficient conditions for conflict-free computations based on the Hermite normal form of T (Shang and Fortes, 1992). However, necessary *and* sufficient conditions for conflict-free mappings are much harder to derive and remain an open problem when $m < n-2$. Instead, a procedure reported in our previous work (Yang *et al.*, 1992) can be used to test for computational conflicts. This procedure is based on the fact that it is possible to reduce the problem of conflict detection to that of checking if a convex polyhedron contains integral points.

4. Time-Optimal Mappings without Computational Conflicts

We now present two different approaches for selecting optimal time mappings Π given a space allocation matrix S. In other words, we show how to schedule the computations of an algorithm after they have been allocated

to processors. One approach employs a method we have developed earlier (Li and Wah, 1985; O'Keefe *et al.*, 1991; O'Keefe and Fortes, 1986) to intelligently search a solution space in an efficient manner. A second method uses integer linear programming augmented with heuristics, which we illustrate in this section using as an example the matrix-multiplication algorithm. We now briefly discuss the first approach and explain in more detail in Sections 5 and 6 when we present the parameter method.

The fact that the execution time of a schedule Π is a monotonic function of the absolute values of the entries of Π can be used to devise an efficient search of the solution space. The basic idea is to enumerate all the possible values of Π in increasing order of the sum of the absolute values of its entries (this assumes the index-set bounds are the same for every dimension; simple modifications can be made to deal with the general case [Shang and Fortes, 1992]). This search method guarantees that the first feasible solution is optimal because of the monotonic increase in execution time with increasing absolute values of the entries of Π. By feasible, we mean that T satisfies the conditions of causality, routability, freedom of computational conflicts, and rank compatibility. As discussed in the previous subsection, freedom of computational conflicts can be easily tested when T is $(n-1) \times n$ in $O(n)$ time. In the general case this method has complexity $\Theta((2N+1)^n)$, where $N = \min\{N_j - l_j : j = 1, \ldots, n\}$. More efficient search methods may make use of the necessary conditions provided in the theorems in the last subsection. We have studied several techniques of reducing the search complexity (Yang *et al.*, 1992). Examples include starting the search at the lower bound of the sum of the absolute values of Π instead of when the sum is one.

The problem of selecting an optimal schedule for the case $T \in \mathbf{Z}^{(n-1)\times n}$ can be formulated as an integer programming problem as follows.

$$\min f = \sum_{j=1}^{n} |\pi_j|(N_j - l_j) \tag{4.1}$$

$$\text{subject to} \begin{cases} (1)\ \Pi D > \vec{0} \\ (2)\ SD = PK \text{ and } \sum_{i=1}^{w} k_{i,j} \le \Pi \vec{d}_j,\ j = 1, \ldots, r \\ (3)\ \text{existing } j \in \{1, \ldots, n\},\ |f_j(\pi_1, \ldots, \pi_n)| > N_j - l_j \\ (4)\ \Pi \in \mathbf{Z}^{1\times n} \end{cases} \tag{4.2}$$

where

$$T = \begin{bmatrix} S \\ \Pi \end{bmatrix},$$

S and P are given, and $f_j, j = 1, \ldots, n$, are as defined in (3.2). As discussed previously, constraint 3 guarantees freedom of computational conflicts and implies that $rank(T) = m + 1$. Constraint 2 is not required if a new processor array is specially designed for the algorithm, or yields linear constraints if P is known. Constraint 3 in (4.2) is linear because $T \in \mathbf{Z}^{(n-1)\times n}$, i.e., the dimension of the algorithm is reduced by one. The formulation in (4.1) and (4.2) is, therefore, an integer piecewise linear programming problem if, as is in the next example (4.1), constraint 1 in (4.2) requires $\pi_j > 0, j = 1, \ldots, n$. This relaxes the absolute-value requirement in the objective function in (4.1). Further, this integer piecewise linear programming problem can be converted to a piecewise linear programming problem for some applications, as illustrated in Example 4.1.

We can add one more constraint, that $gcd(f_1, \ldots, f_n) = 1$, where f_j, $j = 1, \ldots, n$, as defined in (3.2), to the formulation in (4.1) and (4.2) to guarantee that the greatest common divisor of the resulting conflict vector will be unity. However, this makes the problem more difficult to solve. Hence, we ignore this constraint and check the feasibility of the conflict vector of the resulting solution after the solution has been found. In other words, the conflict vector may not be feasible after the common factor of its entries is removed (Shang and Fortes, 1992).

In general, integer programming problems are NP-complete (Schrijver, 1986, p. 245). However, there are two approaches in which this optimization problem may be solved efficiently. First, for each fixed natural number n, there exists a polynomial-time algorithm that solves the optimization problem in time that is a polynomial function of the number of constraints and a logarithmic function of the problem-size variables $N_j - l_j$, $j = 1, \ldots, n$ (Schrijver, 1986, p. 259). Since in our case, n, the dimension of the recurrence equation, and the number of constraints are relatively small, the optimization problem formulated in (4.1) and (4.2) can be solved efficiently. Second, given that the objective function is convex, the optimal solution to the integer linear programming problem of (4.1) and (4.2) is the same as that of the corresponding linear programming problem (with integrality constraints removed) if the solution at the extreme points are integral. This is the method we have used in finding the optimal solution in the following example.

Example 4.1. Consider the matrix-multiplication algorithm and space allocation matrix $S = [1, -1, 0]$. Its dependency matrix D and index set J are shown in (1.10). To satisfy constraint 1 in (4.2), each entry of the linear schedule vector Π must be positive, i.e., $\pi_j \geq 1, j = 1, \ldots, 3$. Therefore, the problem of finding an optimal linear schedule vector for the matrix-multiplication algorithm is formulated as an integer piecewise linear

programming problem:

$$\min f = N(\pi_1 + \pi_2 + \pi_3)$$

$$\text{subject to} \begin{cases} (1)\ \pi_j \geq 1, j = 1, 2, 3 \\ (2)\ SD = PK \text{ and } \sum_{i=1}^{w} k_{i,j} \leq \Pi \vec{d}_j, j = 1, \ldots, 3 \\ (3)\ \pi_3 \geq N, \text{ or } \pi_1 + \pi_2 \geq N \\ (4)\ \Pi \in \mathbf{Z}^{1\times 3} \end{cases} \tag{4.3}$$

where the inequalities in constraint 3 are derived in Example 3.1 and shown in (3.3).

A linear systolic array is to be designed specially for the matrix-multiplication algorithm. Thus, constraint 2 in (4.3) can be ignored. Actually, if one insists on having near-neighbor connections, constraint 2 yields the constraints $\pi_1 \geq 1$, $\pi_2 \geq 1$, and $\pi_3 \geq 0$. This is true because if P and K are chosen as $[1, -1, 0]$ and I (the identity matrix), respectively, $SD = SI = [1, -1, 0] = PK = [1, -1, 0]I$. In fact, these constraints are subsumed by constraint 1. For an integer linear programming problem with convex solution set, if all its extreme points are integral, one of the extreme points is the optimal solution of that problem (Schrijver, 1986, p. 232]. The solution set of the integer programming problem in (4.3) is not convex because of constraint 3, although all the extreme points are integral. One way of solving this problem is to partition the solution set into two disjoint convex subsets and find all the local optimal solutions for all the disjoint solution subsets. If the local optimal solution with the smallest value of the objective function is satisfactory, it is the optimal solution of the integer programming problem in (4.3). ■

The integer piecewise linear programming problem in (4.3) can be decomposed into two integer linear programming subproblems as follows:

$$(I)\ \min f = N(\pi_1 + \pi_2 + \pi_3) \tag{4.4a}$$

$$\text{subject to} \begin{cases} (1)\ \pi_j \geq 1, j = 1, 2, 3 \\ (2)\ \pi_3 \geq N \\ (3)\ \pi_1 + \pi_2 \leq N \\ (4)\ \Pi \in \mathbf{Z}^{1\times 3} \end{cases}$$

$$(II)\ \min f = N(\pi_1 + \pi_2 + \pi_3) \tag{4.4b}$$

$$\text{subject to} \begin{cases} (1)\ \pi_j \geq 1, j = 1, 2, 3 \\ (2)\ \pi_1 + \pi_2 \geq N \\ (3)\ \Pi \in \mathbf{Z}^{1\times 3} \end{cases}$$

Each of these problems is an integer linear programming problem with convex solution set. We can check that every extreme point of these convex sets is integral. Each extreme point is the solution of three of the following five equations: $\pi_1 = 1$, $\pi_2 = 1$, $\pi_3 = 1$, $\pi_3 = N$, and $\pi_1 + \pi_2 = N$. There are five such solutions from these five equations that satisfy $\Pi D > \vec{0}$ as follows: $\Pi_1 = [1, 1, N]$, $\Pi_2 = [1, N-1, 1]$, $\Pi_3 = [1, N-1, N]$, $\Pi_4 = [N-1, 1, 1]$, and $\Pi_5 = [N-1, 1, N]$. The extreme points with the shortest execution time are Π_2 and Π_4. The conflict vectors for Π_2 and Π_4 are, according to (3.3), $[1, 1, -N]^T$, which is feasible because the absolute value of the third entry of the conflict vector is greater than the corresponding size $N - 1$. So both Π_2 and Π_4 are feasible and optimal because their conflict vectors are feasible, and they have the shortest execution time. If we choose Π_2, the total execution time is $t = (N-1)(1+N) + 1 = N^2$ according to (2.4), and $N - 2$ buffers are needed between the two PEs on the link of data A induced by the dependency $\vec{d}_2$, since

$$\Pi_2 \vec{d}_2 - \sum_{j=1}^{3} k_{j2} = N - 1 - 1 = N - 2.$$

Figure 2 shows the block diagram of the linear array for multiplying two 4×4 matrices ($N = 4$). Figure 3 shows the execution of the matrix-multiplication algorithm for the corresponding mapping matrix

$$T = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 3 & 1 \end{bmatrix}.$$

The computation $c_{i_1, i_2} = c_{i_1, i_2} + a_{i_1, i_3} \cdot b_{i_3, i_2}$ indexed by $\vec{I} = [i_1, i_2, i_3]^T$ is executed at processor $[1, -1, 0]\vec{I}$ and at time $[1, 3, 1]\vec{I}$. By inspecting Fig. 2, we can confirm that there are no computational conflicts. Two buffers are needed between the two PEs on the link for data A, or for dependency vector $\vec{d}_2$. The total execution time is 16, and the total number of PEs is 7. As shown in Fig. 2, two data links are used, one for data A traveling from left to right and one for data B traveling from right to left. Data C are stationary and PE_i, $-3 \le i \le 3$, computes c_{i_1, i_2} such that $i_1 - i_2 = i$. (For example, PE_0 computes $c_{1,1}$, $c_{2,2}$, $c_{3,3}$, and $c_{4,4}$.)

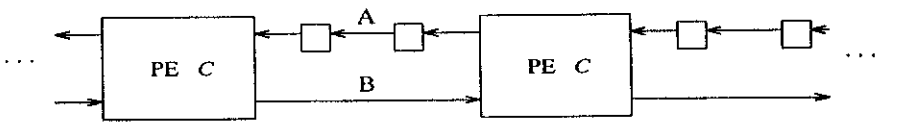

FIG. 2. Block diagram of the linear array for matrix multiplication.

Time	PE-3	PE-2	PE-1	PE0	PE1	PE2	PE3
5				1 a11 1 b11 1 c11			
6				1 a12 1 b21 2 c11	2 a21 1 b11 1 c21		
7				1 a13 1 b31 3 c11	2 a22 1 b21 2 c21	3 a31 1 b11 1 c31	
8			1 a11 2 b12 1 c12	1 a14 1 b41 4 c11	2 a23 1 b31 3 c21	3 a32 1 b21 2 c31	4 a41 1 b11 1 c41
9			1 a12 2 b22 2 c12	2 a21 2 b12 1 c22	2 a24 1 b41 4 c21	3 a33 1 b31 3 c31	4 a42 1 b21 2 c41
10			1 a13 2 b32 3 c12	2 a22 2 b22 2 c22	3 a31 2 b12 1 c32	3 a34 1 b41 4 c31	4 a43 1 b31 3 c41
11		1 a11 3 b13 1 c13	1 a14 2 b42 4 c12	2 a23 2 b32 3 c22	3 a32 2 b22 2 c32	4 a41 2 b12 1 c42	4 a44 1 b41 4 c41
12		1 a12 3 b23 2 c13	2 a21 3 b13 1 c23	2 a24 2 b42 4 c22	3 a33 2 b32 3 c32	4 a42 2 b22 2 c42	
13		1 a13 3 b33 3 c13	2 a22 3 b23 2 c23	3 a31 3 b13 1 c33	3 a34 2 b42 4 c32	4 a43 2 b32 3 c42	
14	1 a11 4 b14 1 c14	1 a14 3 b43 4 c13	2 a23 3 b33 3 c23	3 a32 3 b23 2 c33	4 a41 3 b13 1 c43	4 a44 2 b42 4 c42	
15	1 a12 4 b24 2 c14	2 a21 4 b14 1 c24	2 a24 3 b43 4 c23	3 a33 3 b33 3 c33	4 a42 3 b23 2 c43		
16	1 a13 4 b34 3 c14	2 a22 4 b24 2 c24	3 a31 4 b14 1 c34	3 a34 3 b43 4 c33	4 a43 3 b33 3 c43		
17	1 a14 4 b44 4 c14	2 a23 4 b34 3 c24	3 a32 4 b24 2 c34	4 a41 4 b14 1 c44	4 a44 3 b43 4 c43		
18		2 a24 4 b44 4 c24	3 a33 4 b34 3 c34	4 a42 4 b24 2 c44			
19			3 a34 4 b44 4 c34	4 a43 4 b34 3 c44			
20				4 a44 4 b44 4 c44			

FIG. 3. Execution of multiplication of two 4×4 matrices $C = A \times B$. The small block with leftmost column $[i_1, i_2, i_3]^T$ corresponds to the computation $c_{i_1,i_2} = c_{i_1,i_2} + a_{i_1,i_3} \cdot b_{i_3,i_2}$, which is executed at PE $i_1 - i_2$ and at time $i_1 + 3i_2 + i_3$.

The method discussed here does not guarantee absence of conflicts in data communication over the same link at the same time. We assume that there is enough bandwidth (through hardware links or virtual channels) between the communicating processors to support all the necessary data transfers. Alternatively, if data conflicts must be avoided, one must check the resulting designs for their occurrence. The designs obtained above have no data collisions if data can start to flow at any processor (or data do not have to enter the array solely from the leftmost or the rightmost processor), and data stop flowing as soon as they are no longer needed. This is true because in every column and every row of the matrix K there is only one nonzero entry $k_{j,j} = 1$, $j = 1, \ldots, 3$. This means that when data pass from the source to the destination, they use the data link just once (one hop between source and destination).

Data-link collisions may occur if the data use links more than once when passing from the source to the destination. For example, if the space allocation matrix $S' = [1, 1, N]$ and $P' = [1, 1, 1]$, to satisfy the condition $SD = PK$, one possible set of values for K is $k_{1,1} = k_{2,2} = 1$, $k_{3,3} = N$, and $k_{i,j} = 0$, $i \neq j$. Thus, the distance between the source and destination for data C is N PEs and data C will take N hops over the third link in the processor array, or the link for C to reach the destination. Suppose PE_j, $j = 1, \ldots, N$, are sending data $x_{i,j}$ (corresponding to $c_{i,j}$ of matrix C) to PE_{j+N} at time t_i, $i = 1, \ldots, N$. Then at time t_1, $x_{1,1}$ is on the link between PE_1 and PE_2. At time t_2, two pieces of data $x_{1,1}$ and $x_{2,2}$ are on the link between PE_2 and PE_3, and so on. At time $t_N - 1$, $N - 1$ pieces of data $x_{1,1}, x_{2,2}, \ldots, x_{N-1,N-1}$ are on the link between PE_{N-1} and PE_N. So link collisions exist after time t_1. This is caused by $k_{3,3} = N$. As shown in Fig. 3, there is no link collision for the particular case $N = 4$ illustrated above.

5. Parameter-Based Methods

In the previous section, we described a dependency-based approach (DM) for mapping algorithms to processor arrays. The approach is general and can synthesize processor arrays for algorithms with uniform as well as nonuniform recurrences. In this approach, a desired mapping can be found by determining the elements of a transformation matrix T. Since these elements have to be integers, finding an optimal design requires, in the general case, solving at least an integer linear programming problem. To reduce the complexity, the allocation matrix S can first be chosen heuristically, after which an optimal schedule vector Π is found. For instance, an allocation matrix that uses a small number of processing

elements can be used, and a design that minimizes the completion time can then be obtained on the basis of the matrix.

A more efficient design can be found if the designs are restricted to the case of recurrences with uniform indexing functions. In the next two sections, we present a parameter-based approach for mapping such recurrences. The thinking behind this method is as follows. It is known that the semantics of systolic arrays can be formally described by uniform recurrence equations, i.e., systolic arrays are *isomorphic* to uniform recurrences. This implies that as long as the computations defined by the uniform recurrences are well-formed, there is a direct mapping from the recurrence to the systolic array. In fact, this mapping is equivalent to a *linear* transformation of the index set. Hence, for a linear mapping, the time (respectively, the distance) is constant between execution of any two points $\vec{I}_1$ and $\vec{I}_2$ in the index set separated by a dependence vector $\vec{d}$, where $\vec{I}_1 = \vec{I}_2 + \vec{d}$. This constant is equal to $\Pi\vec{d}$ (respectively, $S\vec{d}$) independent of the index points $\vec{I}_1$ and $\vec{I}_2$. For recurrences with uniform indexing functions (i.e., uniform recurrences and uniformized linear recurrences), the dependences are constant vectors and homogeneous (i.e., the set of dependency vectors at any one point in the index set is the same as at any other point in the index set). Thus, the computation of the recurrence on the processor array is periodic in time and space along the dependency directions in the index space. This periodicity is succinctly captured and exploited in the parameter-based approach that we shall discuss in the balance of this paper. In other words, parameter-based methods employ a different representation that captures the above periodicity, making it possible to find the optimal target array in an efficient manner.

Work on parameter-based methods was first done by Li and Wah (1985) for a restricted set of uniform recurrences. They considered, in particular, three- and two-dimensional recurrences and mapped them to two- and one-dimensional arrays, respectively. The structure of the recurrence was such that the dependency vectors were unit vectors and the dependency matrix an identity matrix. This was an important initial step in obtaining optimal processor arrays efficiently.

This array-synthesis technique using parameters was considerably extended and generalized subsequently into a general parameter method (GPM) (Ganapathy and Wah, 1992a,b). Here the recurrence model was a general n-dimensional recurrence instead of a specific three-dimensional recurrence. The target arrays are also permitted to be of any lower dimension m (where $m < n$). It is assumed that the processing elements are equally spaced in m dimensions with unit distance between directly connected processing elements; buffers between directly connected processing elements, if any, are assumed to be equally spaced along the link.

5.1. Parameters

In GPM, the characterization of the behavior, correctness, and performance of a systolic array is defined in terms of a set of scalar and vector parameters. The crux of GPM is the characterization of the behavior, correctness, and performance of a systolic array by a set of vector and scalar parameters. When a uniform recurrence is executed on a systolic array, the computations are periodic and equally spaced in the systolic array. GPM captures this periodicity by a minimal set of parameters, which is defined as follows.

Parameter 1: Periods. The periods capture the time between execution of the source and sink index points of a dependency vector. Suppose that the time at which an index point $\vec{I}$ (defined for the uniform recurrence equation) is executed is given by a function $\tau_c(\vec{I})$, and let the period of computation t_j along the dependency direction $\vec{d}_j$ be defined as follows:

$$t_j = \tau_c(\vec{I} + \vec{d}_j) - \tau_c(\vec{I}), \qquad j = 1, 2, \ldots, r. \tag{5.1}$$

The number of periods defined is equal to r, the number of dependencies in the algorithm. In terms of DM, period t_j satisfies the following equation:

$$t_j = \Pi \vec{d}_j, \tag{5.2}$$

where Π is the schedule vector in DM.

Parameter 2: Velocity. The velocity of a datum is defined as the directional distance traversed in a single clock cycle; it is denoted $\vec{V}_j$. Since each PE is at unit distance from each neighbor, and buffers (if present) must be equally spaced between pairs of PEs, the magnitude of the velocity vector must be a rational number of the form i/j where i, j are integers and $i \leq j$ (to prevent broadcasting).[3] This implies that in j clock cycles, x propagates through i PEs and $j - i$ buffers. All tokens of the same variable have the same velocity (both speed and direction), which is constant during execution in the systolic array. The total number of velocity parameters is r (one for each dependency vector) and each velocity is an m-element vector, where m is the dimension of the processor array. Hence, the velocity $\vec{V}_j$ is given by

$$\vec{V}_j = \frac{\vec{k}_j}{t_j}, \qquad j = 1, 2, \ldots, r, \tag{5.3}$$

where $\vec{k}_j$ is the (vector) distance between the execution locations of the source and sink index points of the dependency vector $\vec{d}_j$. In the notation

[3] A vector is characterized by its magnitude and a unit directional vector.

of DM, S, the allocation matrix, is related to $\vec{k}_j$ and $\vec{d}_j$ as follows:

$$\vec{k}_j = S\vec{d}_j. \tag{5.4}$$

Parameter 3: Spacing or data distribution. Consider a variable Ω_i pipelined along the dependence vector $\vec{d}_i$, $1 \le i \le r$. The token $\Omega_i(\vec{I} - \vec{d}_i)$ is used at the index points $\vec{I} + t\vec{d}_i$, $t = \ldots, -2, -1, 0, 1, 2, \ldots$, in computing the recurrence. In other words, the token moves through processors that use the variable Ω_i at the index points $(\vec{I} + t\vec{d}_i)$. Consider another token $\Omega_i(\vec{I} - \vec{d}_j)$ of the same variable Ω_i used at index points $(\vec{I} - \vec{d}_j + t\vec{d}_i)$, $j \neq i$. The directional distance in the processor space from token $\Omega_i(\vec{I} - \vec{d}_j)$ to token $\Omega_i(\vec{I} - \vec{d}_i)$ is defined as a spacing parameter[4] $\vec{s}_{i,j}$. Since there are r dependency vectors $\vec{d}_i$, $1 \le i \le r$, there are $r - 1$ nontrivial spacing parameters for each variable and a single trivial spacing parameter $\vec{s}_{i,i} = \vec{0}$. These denote the r distances for variable i: $\Omega_i(\vec{I} - \vec{d}_j) \to \Omega_i(\vec{I} - \vec{d}_i)$, $i, j = 1, 2, \ldots, r$. Each spacing parameter $\vec{s}_{i,j}$ is an m-dimensional vector, where m is the dimension of the processor array. The notation $\vec{s}_{i,j}$ denotes that it is the jth spacing parameter of the ith variable. A total of $r(r-1)$ nontrivial spacing parameters are defined. To compute $\vec{s}_{i,j}$, consider the movement of token $\Omega_j(\vec{I} - \vec{d}_j)$ of variable Ω_j from index point $(\vec{I} - \vec{d}_j)$ to index point $\vec{I}$ with velocity $\vec{v}_j$. In the notation of DM (based on (5.3) and (5.4) and Theorem 5.1),

$$\vec{s}_{i,j} = \vec{v}_j t_j - \vec{v}_i t_j = \vec{k}_j - \vec{v}_i t_j = S\vec{d}_j - \frac{\vec{k}_i}{t_i} t_j = S\vec{d}_j - \frac{\Pi\vec{d}_j}{\Pi\vec{d}_i} S\vec{d}_i. \tag{5.5}$$

The total number of parameters defined is $r \times (r + 2)$, of which r are periods (scalars); the remaining $r^2 + r$ parameters are m-dimensional vectors, of which r are velocities and r^2 are spacings (and r of these spacings are trivially zero).

Example 5.1. Consider a three-dimensional recurrence with $n = 3$, $r = 5$,

$$\begin{aligned} \mathcal{Z}(k, i, j) = {} & X(k, i)Y(j, k) + \mathcal{Z}(k - 1, i + 1, j + 1) \\ & + \mathcal{Z}(k - 1, i + 1, j) + \mathcal{Z}(k - 1, i, j + 1). \end{aligned} \tag{5.6}$$

After pipelining, (5.6) becomes

$$\begin{aligned} \mathcal{Z}(k, i, j) = {} & \mathfrak{X}(k, i, j - 1)\mathfrak{Y}(k, i - 1, j) + \mathcal{Z}(k - 1, i + 1, j + 1) \\ & + \mathcal{Z}(k - 1, i + 1, j) + \mathcal{Z}(k - 1, i, j + 1). \end{aligned} \tag{5.7}$$

[4] Spacing parameters in GPM are denoted by $\mathbb{S}$, whereas the processor-allocation matrix in DM is denoted by $\mathbb{S}$.

Let

$$\vec{I} = (i, j, k)^T, \qquad \vec{d}_1 = (0, 0, 1)^T,$$
$$\vec{d}_2 = (0, 1, 0)^T, \qquad \vec{d}_3 = (1, -1, -1)^T,$$
$$\vec{d}_4 = (1, -1, 0)^T, \qquad \vec{d}_5 = (1, 0, -1)^T.$$

Rewriting the recurrence in the functionally equivalent form,

$$\mathcal{Z}(\vec{I}) = \mathfrak{X}(\vec{I} - \vec{d}_1) \times \mathcal{Y}(\vec{I} - \vec{d}_2) + \mathcal{Z}(\vec{I} - \vec{d}_3),\ \mathfrak{A}_4(\vec{I} - \vec{d}_4),\ \mathfrak{A}_5(\vec{I} - \vec{d}_5), \tag{5.8a}$$

we obtain

$$\mathfrak{A}_i(\vec{I}) = \mathcal{Z}(\vec{I}), \qquad i = 4, 5, \tag{5.8b}$$

where $\mathfrak{A}_i$ are dummy variables. Each dependence is now associated with one variable. The dependence vectors, collected into a matrix, are

$$D = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \\ \mathfrak{X} & \mathcal{Y} & \mathcal{Z} & \mathcal{Z} & \mathcal{Z} \end{bmatrix} \tag{5.9}$$

The parameters defined are as follows. The variables $\mathfrak{X}$, $\mathcal{Y}$, $\mathcal{Z}$, $\mathfrak{A}_4$, and $\mathfrak{A}_5$ have periods t_1, t_2, t_3, t_4, and t_5, respectively, and velocities $\vec{V}_1$, $\vec{V}_2$, $\vec{V}_3$, $\vec{V}_4$, and $\vec{V}_5$, respectively, where

$$\begin{aligned} t_1 &= \tau_c(k, i, j) - \tau_c(k, i, j - 1) \\ t_2 &= \tau_c(k, i, j) - \tau_c(k, i - 1, j) \\ t_3 &= \tau_c(k, i, j) - \tau_c(k - 1, i + 1, j + 1) \\ t_4 &= \tau_c(k, i, j) - \tau_c(k - 1, i + 1, j) \\ t_5 &= \tau_c(k, i, j) - \tau_c(k - 1, i, j + 1). \end{aligned} \tag{5.10}$$

There are 25 spacing parameters $\vec{S}_{i,j}$, $i, j = 1, 2, 3, 4, 5$, with $\vec{S}_{i,i} = \vec{0}$. For instance, consider the spacings of the first variable $\mathfrak{X}$: $\vec{S}_{1,2}$, $\vec{S}_{1,3}$, $\vec{S}_{1,4}$, and $\vec{S}_{1,5}$. These are defined as the distances

$$\left\{ \begin{array}{l} (\mathfrak{X}(k, i, j - 1) \to \mathfrak{X}(k, i - 1, j)), \\ (\mathfrak{X}(k, i, j - 1) \to \mathfrak{X}(k - 1, i + 1, j + 1)), \\ (\mathfrak{X}(k, i, j - 1) \to \mathfrak{X}(k - 1, i + 1, j)), \\ (\mathfrak{X}(k, i, j - 1) \to \mathfrak{X}(k - 1, i, j + 1)) \end{array} \right\},$$

respectively. Using the indexing function of $\mathfrak{X}$, $\vec{s}_{1,2}$, $\vec{s}_{1,3}$, $\vec{s}_{1,4}$, and $\vec{s}_{1,5}$ correspond to the distances

$$\begin{Bmatrix} (X(k,i) \to X(k,i-1)), \\ (X(k,i) \to X(k-1,i+1)), \\ (X(k,i) \to X(k-1,i+1)), \\ (X(k,i) \to X(k-1,i)) \end{Bmatrix},$$

respectively. ■

5.2 Constraints

In Section 5.1, a set of $r^2 + r$ parameters was introduced to define a target systolic array. The assignment of values to these parameters defines a specific systolic array with a particular number of processors, buffers, and data-input patterns. It is also easy to see that all systolic arrays that solve a given algorithm (or uniform recurrence) correspond to some assignment of values to the parameters. Hence, choosing different values for these parameters leads to different array configurations with different performances. As a result, the problem of array design has been reduced to that of choosing appropriate parameter values.

The choice of a value for one of the $r^2 + r$ parameters is not independent of the choice of values for the other parameters. In this section, constraint equations relating the parameters are given such that the set of values for the parameters are meaningful and define a valid systolic array. Theorems 5.1 and 5.2 provide fundamental space-time relationships that must be satisfied by the parameters to ensure correct systolic processing. The avoidance of computational and data-link conflicts is enforced by the condition in Theorem 5.3. The theorems are provided without proofs due to space limitations.

The following notation is introduced to simplify the presentation of the theorems. Let $\vec{T} = \{t_1, t_2, \ldots, t_r\}$ be a vector composed of periods, and let $\mathcal{K} = [\vec{k}_1, \vec{k}_2, \ldots, \vec{k}_r]$ be a matrix (of size $m \times r$, where m is the dimension of the systolic array) composed of displacements $\vec{k}_i = \vec{V}_i t_i$. Both $\vec{T}$ and $\vec{k}_i$ are $r \times 1$ column vectors. The displacement $\vec{k}_i$ is synonymous with the velocity $\vec{V}_i$, because the choice of one immediately determines the other. In searching for parameter values, we choose to consider $\vec{k}_i$ and not $\vec{V}_i$.

Theorem 5.1. *The parameters velocities, spacings, and periods must satisfy the following constraint equations for correct systolic processing:*

$$\vec{V}_i t_i = \vec{V}_j t_i + \vec{s}_{j,i}, \qquad i, j = 1, 2, \ldots, r. \tag{5.11}$$

These constraints ensure that in computing an index point $\vec{I}$ at any processor in the array, all the participating data tokens must be present at the processor at the same time after moving from their respective processors where they had been used earlier. A total of r^2 constraints are obtained from Theorem 5.1.

Let $\mathcal{S} = [\vec{s}_{i,j}]$, $i, j = 1, 2, \ldots, r$, be an $r \times r$ "matrix" (actually, a matrix of vectors) of spacings such that the (i, j)th element of the matrix is $\vec{s}_{i,j}$. Note that, by definition, $\vec{s}_{i,i} = 0$. Let $\mathcal{S}_i$ be the ith "row" of this "matrix" $\mathcal{S}$, i.e., $\mathcal{S}_i = [\vec{s}_{i,1}, \vec{s}_{i,2}, \ldots, \vec{s}_{i,r}]$ (where $\mathcal{S}_i$ is an $m \times r$ matrix). Since $\vec{s}_{i,j} = \vec{V}_j t_j - \vec{V}_i t_j = \vec{k}_j - \vec{V}_i t_j$ from Theorem 5.1, it can be written in matrix form as

$$\mathcal{S}_i = \mathcal{K} - V_i \otimes \vec{T}, \tag{5.12}$$

where $\otimes$ is the outer or tensor product, i.e., $\vec{a} \otimes \vec{b} = \vec{a}\vec{b}^T = [a_i b_j]$.

The next theorem characterizes the constraints on the periods and displacements if the dependencies in the recurrence are not linearly independent.

Let g be the rank of the dependency matrix D. Therefore, $\mathfrak{N}$, the null space of D, has $r - g$ columns (since D has r columns). Let

$$\mathfrak{N} = [\vec{\alpha}_1 \ \vec{\alpha}_2 \ \ldots, \ \vec{\alpha}_{r-g}]$$

be an $r \times (r - g)$ matrix, where $\vec{\alpha}_i$, $i = 1, 2, \ldots, (r - g)$, are the basis vectors of the null space of D. Hence,

$$D \cdot \vec{\alpha}_i = 0, \qquad 1 \le i \le (r - g). \tag{5.13}$$

Theorem 5.2. *The periods t_i and the displacements $\vec{k}_i$ are related as follows:*

$$\vec{T} \cdot \mathfrak{N} = 0, \tag{5.14}$$

$$\mathcal{K}\mathfrak{N} = 0, \tag{5.15}$$

where $\mathfrak{N}$ is a matrix consisting of the basis vectors of the null space of D.

The implication of Theorem 5.2 is as follows. If the dependency matrix D is not full rank, i.e., some of the column vectors, say $\vec{d}_j$ of D, can be written as linear combinations of other column (dependency) vectors, the periods of computation t_j (respectively, displacements $\vec{k}_j$) along the linearly dependent column vectors can be expressed by the same linear combinations of the other periods (respectively, displacements).

Thus, the conditions provided by Theorem 5.2 are additional constraints from the definition of parameters and the dependency vectors in the algorithm. Theorem 5.2 provides a total of $2(r - g)$ constraints.

The following corollary can be easily derived from Theorem 5.2. The implication of this corollary is that only $g - 1$ of the r spacing parameters for each variable are independent, one of them is zero, and the rest can be obtained as linear combinations of the $g - 1$ independent ones.

Corollary 5.1. *The spacing parameters* $\mathbb{S}_i = [\vec{S}_{i,1} \cdots \vec{S}_{i,r}]$ *are constrained by the equations*

$$\mathbb{S}\mathfrak{N} = 0, \qquad i = 1, 2, \ldots, r,$$

where $\mathfrak{N}$ *is a matrix consisting of the basis vectors of the Null Space of D.*

Example 5.2. From Theorem 5.1, the constraint equations for the recurrence in (5.6) (excluding the trivial constraint $\vec{V}_1 t_1 = \vec{V}_1 t_1 + \vec{S}_{1,1}$) are

$$\vec{V}_1 t_1 = \vec{V}_2 t_1 + \vec{S}_{2,1} = \vec{V}_3 t_1 + \vec{S}_{3,1} = \vec{V}_4 t_1 + \vec{S}_{4,1} = \vec{V}_5 t_1 + \vec{S}_{5,1}. \quad (5.16)$$

Similarly, there are 16 additional equations related to $V_2 t_2$, $V_3 t_3$, $V_4 t_4$, and $V_5 t_5$.

D defined in (5.9) has rank 3. Hence, $\mathfrak{N}$ comprises two basis vectors.

$$\mathfrak{N} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad (5.17)$$

From Theorem 5.2, the additional constraints are

$$\begin{aligned} t_4 &= t_1 + t_3 & t_5 &= t_2 + t_3 \\ \vec{k}_4 &= \vec{k}_1 + \vec{k}_3 & \vec{k}_5 &= \vec{k}_2 + \vec{k}_3 \end{aligned} \quad (5.18)$$

In this example, there are a total of 27 vector constraints and two scalar constraints. ■

To summarize, a total of $r^2 + r$ vector parameters and r scalar parameters have been defined whose values have to be determined. Theorems 5.1 and 5.2 give a total of $r^2 + (r - g)$ vector constraints and $(r - g)$ scalar constraints. Hence, g of the scalar parameters (periods) and g of the vector parameters have to be chosen such that the other $r - g$ scalar parameters and the other $r^2 + (r - g)$ vector parameter values can be determined from the chosen scalar and vector constraints. Since the performance of the design can be naturally expressed in terms of periods and displacements, our strategy is to choose the g periods and g displacements and determine the remaining $r - g$ periods and $r - g$ displacements from Theorem 5.2 and all the r^2 spacings

using Theorem 5.1. Corollary 5.1 further states that only $g - 1$ of the r spacings are independent for each variable. All the vector parameters are m-dimensional (with m elements).

The validity of the space-time mapping is governed by the following fundamental necessary and sufficient conditions.

1. *Precedence constraints*. An index point should be executed only after all the index points on which this depends have been executed. In DM, $\Pi D > 0$.
2. *Computational conflicts*. No two index points may be executed at the same processor at the same time. In DM, $\Pi(\vec{I}_1) = \Pi(\vec{I}_2)$ implies that $S(\vec{I}_1) \neq S(\vec{I}_2)$.
3. *Data-link conflicts*. No two data tokens may contend for a given link at the same time.

Having established the parameters and the two basic relationships among them, we show how the fundamental conditions for validity are satisfied in GPM.

By definition, periods denote the time difference between the source and sink of the dependencies. Hence, the precedence constraints are satisfied by simply enforcing $t_i \geq 1$, $i = 1, \ldots, r$. In the array model, all tokens of the same variable move with the same velocity. Hence, data-link conflicts can exist if and only if two tokens of a variable are input at the same time into the same processor and travel together contending for links. This condition is called a *data-input conflict* in GPM, as two data tokens may be in the same physical location and may conflict with each other as they move through the processors together.

It is important to note that in GPM, computational conflicts can exist if and only if data-input conflicts occur. This can be seen by the following simple argument. If two index points are evaluated in the same processor at the same time, then, for each variable, at least two distinct tokens exist together in the same processor. Hence, if there is at least one nonstationary variable, there will be data-input conflict for the tokens of that variable. Otherwise, all the variables are stationary and the entire computation is executed on one processor, i.e., there is no systolic array. Hence, by enforcing a rule that no data-input conflicts exist, both computational and data-link conflicts are avoided. Theorem 5.3 below presents conditions under which data-input conflicts can be eliminated.

Consider the spacings of variable i. Let $\mathbb{S}_i$ be an $m \times (g - 1)$ matrix:

$$\mathbb{S}_i = [\vec{S}_{i,1}, \vec{S}_{i,2}, \ldots, \vec{S}_{i,g-1}], \tag{5.19}$$

where $\vec{S}_{i,1}, \vec{S}_{i,2}, \ldots, \vec{S}_{i,g-1}$ are $g - 1$ consistent spacings. Let $\vec{\alpha}$, $\vec{\beta}$, and $\vec{\gamma}$

be vectors with $g-1$ integral elements. Let L_k, U_k, $k = 1, 2, \ldots, g-1$, be defined such that the position of all the tokens of the input matrix can be represented by $\sum_{k=1}^{g-1} \vec{s}_{i,k}\beta_k$, where $L_k \le \beta_k \le U_k$. L_k and U_k are functions of the size of the input matrix.

Theorem 5.3. *Data-input conflicts* occur *in the input matrix of a nonstationary input i if and only if* $\mathbb{S}_i'\vec{\alpha} = \vec{0}$ *and* $\vec{\alpha} \neq \vec{0}$, *where* $\vec{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_{g-1}]^T$ *and* $\alpha_i \in [(L_i - U_i), \ldots, (L_i + U_i)]$, *for all i such that* $1 \le i \le g-1$.

Proof. The position of any element of input i can be described as $\mathbb{S}_i'\vec{\beta}$, where $\vec{\beta} = [\beta_1, \ldots, \beta_{g-1}]$ and $L_i \le \beta_i \le U_i$. Therefore,

Data-input conflicts

$$\leftrightarrow \mathbb{S}_i'\vec{\beta} = \mathbb{S}_i'\vec{\gamma}, \vec{\beta} \neq \vec{\gamma} \quad \text{and} \quad L_i \le \vec{\gamma}_i, \beta_i \le U_i$$

$$\leftrightarrow \mathbb{S}_i'(\vec{\beta} - \vec{\gamma}) = \vec{0}$$

$$\leftrightarrow \mathbb{S}_i'\vec{\alpha} = \vec{0}, \vec{\alpha} = \vec{\beta} - \vec{\gamma}, \alpha_i \in [(L_i - U_i), \ldots, (L_i + U_i)], \vec{\alpha} \neq \vec{0} \quad \blacksquare$$

Note that in Theorem 5.3, we have defined conservative bounds on α_i. Better estimates can be obtained (Xue, 1993) and will result in less overhead when the conditions in Theorem 5.3 are checked in the design process.

Example 5.3. For the recurrence in (5.6), if the array sought is one-dimensional, the spacing parameters are all one-dimensional scalars. Let $\vec{s}_{1,2}$ and $\vec{s}_{1,5}$ be the two independent spacings for input X. We set the values of L_1 and L_2 to be 1, and the values of U_1 and U_2 to be N. Therefore, according to Theorem 5.3, data-input conflicts occur in input X if and only if

$$[\vec{s}_{1,2}\ \vec{s}_{1,5}]\begin{bmatrix}\alpha_1\\ \alpha_2\end{bmatrix} = 0, \tag{5.20}$$

where $-(N-1) \le \alpha_1, \alpha_2 \le (N-1)$ and $\alpha_1, \alpha_2 \neq 0$. For instance, if $N = 5$ and $\vec{s}_{1,2} = 6$ and $\vec{s}_{1,5} = 4$, we find that $\alpha_1 = 2$ and $\alpha_2 = -3$ satisfies (5.20). (In one dimension, the vector spacings are positive or negative numbers.) Hence, there are data-input conflicts in input X. $\blacksquare$

5.3. Design Method

The design of a feasible processor array is equivalent to choosing an appropriate set of parameters that satisfy the constraints imposed by the dependency and application requirements. The search for the "best" design can be represented by the following optimization problem:

$$\text{Minimize } b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) \tag{5.21}$$

$$\text{Subject to: } \begin{cases} 1 \le t_i,\ i = 1, \ldots, r, \\ 0 \le |\vec{k}_i| \le t_i,\ i = 1, \ldots, r, \\ \text{constraints defined in Theorems 5.1, 5.2, and 5.3,} \\ \#PE \le \#PE^{UB} \text{ and } T_c \le T_c^{UB}. \end{cases}$$

The objective function b defined in (5.21) is expressed in terms of attributes such as T_{comp}, the computation time of the algorithm; T_{load}, the load time for the initial inputs; T_{drain}, the drain time for the final results; and $\#PE$, the number of processing elements in the design. Note that the completion time for evaluating the recurrence is

$$T_c = T_{comp} + T_{load} + T_{drain}. \tag{5.22}$$

All the attributes are then expressed in terms of parameters defined in GPM.

The first two constraints in (5.21) follow directly from the definition of the parameters in GPM. Since the target array is systolic, the displacements $|\vec{k}_i|$ should not exceed the periods t_i in order to prevent data broadcasting (velocities should not exceed one). In addition, the constraints $t_i \ge 1$, $i = 1, 2, \ldots, r$, ensure that the precedence constraints are satisfied.

The third constraint indicates that the recurrence is evaluated correctly by the processor array satisfying the dependency requirements (Theorems 5.1 and 5.2), and is free of data-link and computational conflicts (Theorem 5.3).

The fourth constraint indicates what bounds on T_c and $\#PE$ that are imposed on the design are to be obtained. For instance, the following are two possible formulations of the optimization problem:

(a) Minimize T_c for a design with a maximum bound on $\#PE$ and PE^{UB};
(b) Minimize $\#PE$ for a design with a maximum bound on T_c and T_c^{UB}.

Both of these formulations represent trade-offs between T and $\#PE$. This is a unique advantage to using GPM as a way of synthesizing systolic arrays. Both optimization problems and trade-offs are illustrated in detail in Section 6. Another unique feature of GPM is that the formulation in (5.21) is defined with respect to a specific recurrence and a specific problem size N. This allows a truly application-specific and problem-size-specific systolic array to be designed to suit specific application requirements.

In addition to the constraints we have discussed, there are other constraints that may be defined in the search process. Since, in general, the objective function is nonlinear, involving functions such as ceiling, floor, and the maximum/minimum of a set of terms, it is difficult to describe a

comprehensive algorithm that covers all possible cases. In the following, we first describe our general search strategy, after which we discuss searches with objectives that are functions of T_c, T_{comp}, T_{drain}, and $\#PE$. We then present the search algorithm and show its application to special cases of optimizing T_c and $\#PE$.

Our general search strategy takes the objective function b (assumed to be minimized) and decomposes it into two functions b_1 and b_2 related by f as follows:

$$b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) = f(b_1(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r), b_2(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)), \quad (5.23)$$

where N is not represented explicitly since it is a constant in the optimization. The decomposition is done in such a way that b_1 is a monotonic function of its variables (which are enumerated), and b_2 is a function in which a lower-bound estimate on its value can be obtained easily. In addition, f is assumed to be a monotonically increasing function with increasing values of b_2 so that a lower-bound estimate on b_2 can be used to get an upper bound on b_1. The search proceeds by systematically enumerating all combinations of a selected set of parameters defined in b_1, and solving for the rest of the parameters by the constraints defined in (5.21) or by computing their values when the lower bound of b_2 is evaluated. Every time a combination of parameters in b_1 is searched, a lower-bound estimate on b_2 is computed. This lower-bound estimate, together with $B^{incumbent}$, the objective value of the current incumbent design, defines an upper bound on the value of b_1 to be enumerated further in the search process. That is,

$$B^{incumbent} = f(b_1^{UB}(t_1, \ldots, t_r, \vec{k}_r), b_2^{LB}(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)). \quad (5.24)$$

Note that this equation only defines an upper bound on the value of b_1 to be enumerated; it does not define the combinations of parameter values of b_1 that can be pruned. Pruning of combinations of parameter values of b_1 is possible only if b_1 is monotonic with respect to the combination of parameter values chosen in the enumeration process.

To illustrate our search strategy, consider an objective that is a function of T_{comp}, T_{load}, T_{drain}, and $\#PE$ as follows:

$$B = b_1(T_{comp}, T_{load}, T_{drain}, \#PE) \times b_2(T_{comp}, T_{load}, T_{drain}, \#PE). \quad (5.25)$$

Assume that a lower-bound estimate of b_2 can be obtained by setting $T_{load} = T_{drain} = 0$, $T_{comp} = T_{comp}^{\min}$, and $\#PE = \#PE^{\min}$. Consider a case in which $\#PE$ is expressed as a function of $|\vec{k}_1|, \ldots, |\vec{k}_r|$. $\#PE$ is minimal when exactly one $|\vec{k}_i|$ is 1, and the rest of them $|\vec{k}_j|$, $j \neq i$, are 0. Similarly a crude

$T_{comp}^{\min}$ can be obtained by letting all $t_i = 1$. Hence, given $B^{incumbent}$, we have

$$B^{incumbent} = b_1^{UB}(T_{comp}, T_{load}, T_{drain}, \#PE \mid T_{load} = T_{drain} = 0, \#PE = \#PE^{\min}) \times b_2^{LB}(T'_{comp}, T_{load}, T_{drain}, \#PE \mid T'_{comp} = T_{comp}^{\min}, T_{load} = T_{drain} = 0, \#PE = \#PE^{\min}), \quad (5.26)$$

or, equivalently,

$$T_{comp}^{UB} = b_1^{-1}\left(\frac{B^{incumbent}}{b_2^{LB}(T'_{comp}, T_{load}, T_{drain}, \#PE \mid T'_{comp} = T_{comp}^{\min}, T_{load} = T_{drain} = 0, \#PE = \#PE^{\min})}\right), \quad (5.27)$$

where b_1^{-1} is the inverse function of b_1, and T_{comp} is the dummy parameter T_{comp} used in b_2.

For example, let the objective function be

$$B = (T_{comp} + T_{load} + T_{drain})^2 \times \#PE = b_1(T_{comp} + T_{load} + T_{drain}) \times b_2(\#PE). \quad (5.28)$$

According to (5.27), we have

$$T_{comp}^{UB} = \sqrt{B^{incumbent}/\#PE^{\min}}. \quad (5.29)$$

T_{comp}^{UB} is refined continuously as new incumbent designs are found in the search, and the search stops when there is no combination of t_i, $i = 1, \ldots, r$, that satisfies $T_{comp} \leq T_{comp}^{UB}$.

In the following, we describe the search procedure for an objective function of the form in (5.25).

Search procedure for minimizing $b(\#PE, T_c) = b(T_{comp}, T_{load}, T_{drain}, \#PE)$, where T_{comp} is a function of $t_1, \ldots, t_r$, T_{load} and T_{drain} are functions of $t_1, \ldots, t_r, |\vec{k}_1|, \ldots, |\vec{k}_r|$, and $\#PE$ is a function of $|\vec{k}_1|, \ldots, |\vec{k}_r|$.

1. Choose g periods and g displacements to be unconstrained parameters. Without loss of generality, we may let these periods and displacements be t_i and $\vec{k}_i$, $1 \leq i \leq g$, respectively.
2. Initialize T_{comp}^{UB} to be T_{comp}^{seq}, the computation time required to evaluate the recurrence sequentially.
3. Set the values of all the g unconstrained periods t_i, $i = 1, \ldots, g$, to be unity.
4. Choose the magnitude of g unconstrained displacements $|\vec{k}_i|$, $i = 1, \ldots, g$, to be zero.
5. Compute the values of the other dependent $r - g$ periods and displacements using the conditions of Theorem 5.2.

6. Compute T_{comp}^{cur} using the periods and displacements found, where T_{comp}^{cur} is the computation time (without load and drain times) required for processing the recurrence by substituting the current values of t_i, $i = 1, \ldots, r$. (Note that the design may not be feasible at this time.) If $T_{comp}^{cur} > T_{comp}^{UB}$, exit with the incumbent design.
7. Solve for the spacing parameters from (5.11) defined in Theorem 5.1.
8. Check for data-input conflicts using Theorem 5.3 on the spacing parameters; also, check whether the constraints on T_c and $\#PE$ are violated (constraint 4 in (5.21)).
9. If the solution is not feasible, increment one $|\vec{k}_i|$ and repeat Steps 5, 6, 7, and 8 until $|\vec{k}_i|$ are all equal to t_i, $i = 1, \ldots, r$. If all the $|\vec{k}_i|$ equal t_i and no feasible design is found, go to Step 10. If a feasible design is found, go to Step 11.
10. Increment one of the periods such that T_{comp}^{cur} increases by the lowest possible value. Go to Step 4.
11. Compute B^{cur}, the objective value achieved by the current design found. If $B^{cur} < B^{incumbent}$, set $B^{incumbent} = B^{cur}$ and compute T_{comp}^{UB} for the current design using (5.27). Increment one $|\vec{k}_i|$ and go to Step 5.

The worst-case complexity of the search procedure above is $(T_{comp}^{seq})^{2g}$, where T_{comp}^{seq} is the time needed to process the recurrence sequentially. This bound is true because we iterate in the worst case all combinations of t_i and $|\vec{k}_i| \leq t_i$, $i = 1, \ldots, r$.

A special case of the optimization is to find a design with minimum computation time (not including load and drain times). This is discussed in Section 4 of this paper as well as in our earlier work (Ganapathy and Wah, 1992a,b). In this case, b_2 is a constant function, and b_1 a linear function of $t_1, \ldots, t_r$. Hence, the first feasible design found sets T_{comp}^{UB} equal to T_{comp}^{cur} of the feasible design obtained, and the first feasible design becomes the optimal design that minimizes T_{comp}.

For a design that minimizes $\#PE$, the search procedure described above needs to be changed. In this case, b_1 should be defined as a function of $|\vec{k}_1|, \ldots, |\vec{k}_r|$. The search should start iterating with the smallest combinations of these variables.

6. Applications of the General Parameter Method

Path-finding problems belong to an important class of optimization problems. Typical examples include computing the transitive closure and the shortest paths of a graph. Two-dimensional systolic arrays for finding transitive closures have been studied extensively in the literature (Kung *et al.*,

1987; Guibas *et al.*, 1979; Rote, 1985). In this section we synthesize a one-pass linear systolic array for the Warshall–Floyd path-finding algorithm. The discussion below is with respect to the transitive closure problem.

The transitive closure problem is defined as follows. Compute the transitive closure $C^+[i,j]$ of an n-node directed graph with an $n \times n$ Boolean adjacency matrix $C[i,j]$, where $C[i,j] = 1$ if there is an edge from vertex i to vertex j or $i = j$, and $C[i,j] = 0$ otherwise. Since the dependency structure is irregular and difficult to map, S. Y. Kung *et al.* (1987) converted the transitive closure algorithm into a reindexed form and mapped it to 2-D spiral and orthogonal arrays. Based on their algorithm we obtain the following five dependency vectors after pipelining the variables:

$$\begin{aligned}
\vec{d}_1 &= (0,0,1)^T \text{ for } (k,i,j)^T \to (k,i,j-1)^T,\ 2 \le j \le N,\\
\vec{d}_2 &= (0,1,0)^T \text{ for } (k,i,j)^T \to (k,i-1,j)^T,\ 2 \le i \le N,\\
\vec{d}_3 &= (1,-1,-1)^T \text{ for } (k,i,j)^T \to (k-1,i+1,j+1)^T,\\
&\qquad 2 \le k \le N,\ 1 \le i,j \le N-1, \qquad (6.1)\\
\vec{d}_4 &= (1,-1,0)^T \text{ for } (k,i,N)^T \to (k-1,i+1),N)^T,\\
&\qquad 2 \le k \le N,\ 1 \le i \le N-1,\\
\vec{d}_5 &= (1,0,-1)^T \text{ for } (k,N,j)^T \to (k-1,N,j+1)^T,\\
&\qquad 2 \le k \le N,\ 1 \le j \le N-1,
\end{aligned}$$

where $\vec{I}_1 \to \vec{I}_2$ means that the data at point $\vec{I}_2$ is used at point $\vec{I}_1$. For nodes on the boundary of dependency graph G where $i = N$ (respectively, $j = N$), dependency $\vec{d}_4$ (respectively, $\vec{d}_5$) is present instead of dependency $\vec{d}_3$. For other interior points, only the three dependencies $\vec{d}_1$, $\vec{d}_2$, and $\vec{d}_3$ exist.

The running example discussed in Section 5 is a recurrence with the five dependencies listed above. The dependency graph of the recurrence used in example [(5.6)] is regular and homogeneous with five dependencies at each point. However, for transitive closure the dependency graph is not completely regular. Hence, control bits are used to modify the flow (or velocity) of the tokens in order to execute the dependency graph on the processor array correctly.

The key observation is as follows. Matrix C (whose transitive closure is to be found) is input along dependency direction $\vec{d}_3$. Inputs along other dependency directions $\vec{d}_1$, $\vec{d}_2$, $\vec{d}_4$, $\vec{d}_5$ are nonexistent, i.e., they are never sent into the array from the external host. Hence, there are no data-input conflicts along these dependency directions as the generated outputs are sent at most once on each link in every cycle of the array. As a result, we need to consider only data-input conflicts along direction $\vec{d}_3$. Since dependencies $\vec{d}_3$, $\vec{d}_4$, and $\vec{d}_5$ never coexist, there are only two spacings for data along direction $\vec{d}_3$, namely, $\vec{s}_{3,1}$ and $\vec{s}_{3,2}$.

A total of eight relevant parameters are defined for the transitive closure problem: three periods t_1, t_2, t_3, three displacements $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3$, and two spacings $\vec{s}_{3,1}$ and $\vec{s}_{3,2}$. For a linear array all the parameters are scalars. Applying Theorem 5.2 and in the same way as in the derivation of (5.18), the periods along directions $\vec{d}_4$ and $\vec{d}_5$ are given as $t_4 = t_1 + t_3$ and $t_5 = t_2 + t_3$. Similarly, the displacements $\vec{k}_4 = \vec{k}_1 + \vec{k}_3$ and $\vec{k}_5 = \vec{k}_2 + \vec{k}_3$. From Theorem 5.1 and (5.3), we get

$$\vec{s}_{3,1} = \frac{t_3\vec{k}_1 - t_1\vec{k}_3}{t_3}, \qquad \vec{s}_{3,2} = \frac{t_3\vec{k}_2 - t_2\vec{k}_3}{t_3}.$$

We illustrate in the rest of this section five formulations of the optimization of systolic arrays: (a) T_{comp}-optimal designs without bound on $\#PE$; (b) T_c-optimal designs without bound on $\#PE$; (c) $\#PE$-optimal designs without bound on T_c or T_{comp}; (d) optimal designs with specific bounds on T_{comp} or $\#PE$; and (e) optimal designs with specific bounds on T_c or $\#PE$. Recall from Section 5 that $T_c = T_{load} + T_{comp} + T_{drain}$, and that we need to express T_{load}, T_{comp}, T_{drain}, and $\#PE$ in terms of the parameters defined in GPM. For this example, T_{comp} and $\#PE$ are stated below without proof (Ganapathy and Wah, 1992b).

Lemma 6.1. *T_{comp}, the computation time (without load and drain times), and $\#PE$, the number of processing elements, for computing an $N \times N$ transitive closure in a linear systolic array satisfying the dependencies defined in* (6.1) *are given by*

$$T_{comp} = (N-1)(2t_1 + 2t_2 + t_3) + 1, \tag{6.2}$$

$$\#PE = (N-1)(|\vec{k}_1| + |\vec{k}_2| + |\vec{k}_1 + \vec{k}_2 + \vec{k}_3|) + 1. \tag{6.3}$$

Due to space limitations, we state below without proof the equations for T_{load} and T_{drain}. The idea behind the proof is to enumerate all the eight possible directions of $\vec{k}_1$, $\vec{k}_2$, and $\vec{k}_3$ and compute the load and drain times for each.

Lemma 6.2. *T_{load}, the load time, and T_{drain}, the drain time, for computing an $N \times N$ transitive closure in a linear systolic array satisfying the dependencies defined in* (6.1) *are given by*

$$T_{load} = T_{drain} = (N-1)\left[\frac{t_3\{g(\vec{k}_1, \vec{k}_3) + g(\vec{k}_2, \vec{k}_3) + g[(\vec{k}_1 + \vec{k}_2 + \vec{k}_3), \vec{k}_3]\}}{|\vec{k}_3|}\right] + 1,$$

$$+ (N-1)\left[\frac{g(\vec{s}_{3,1} - \vec{k}_3) + g(\vec{s}_{3,2} - \vec{k}_3)}{|\vec{v}_3|}\right] \tag{6.4}$$

where

$$g(\vec{x}, \vec{y}) = \begin{cases} |\vec{x}| & \text{if } \vec{x} \text{ and } \vec{y} \text{ are in opposite directions} \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

For linear-array synthesis, since the spacings are scalars, let $s_{3,1} = |\vec{s}_{3,1}|$ and $s_{3,2} = |\vec{s}_{3,2}|$. The condition for data-input conflict (Theorem 5.3) can be refined as given in Theorem 6.1, which we state without proof.

Theorem 6.1. *Data-input conflicts occur in an input matrix C if and only if*

$$\frac{s_{3,1}}{\xi i} < N \quad \text{and} \quad \frac{s_{3,2}}{\xi} < N, \tag{6.6}$$

where $\xi = GCD(s_{3,1}, s_{3,2})$ *and* $GCD(a, b)$ *is the greatest common divisor of* a *and* b.

Table 1 shows the optimal linear designs found by the search procedure of GPM in which the objective is to minimize either T_{comp} of T_c. In finding these designs, t_3 is incremented before t_1 or t_2 in Step 10 of the search procedure presented in Section 5.3 (refer to (6.2)), since such a procedure increases T_{comp} by the least amount.

The designs in the left half of Table 1 are based on first optimizing T_{comp}. From the set of designs that have minimal T_{comp}, we found designs that require the minimal #*PE*, after which we found designs that require minimal T_{load} and T_{drain}. We list T_{load}, T_{comp}, T_{drain}, the #*PE*s needed, and the CPU time used by the search procedure running on a Sun Sparcstation 10/30. The designs found are identical to the designs we have published before (Ganapathy and Wah, 1992a,b), except for the case $N = 8$. In our previous design for $N = 8$ (Ganapathy and Wah, 1992a,b), we had $(t_1, t_2, t_3) = (1, 2, 3)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -2, 1)$. This design was found without considering T_{load} and T_{drain}. As a result, it requires $T_{load} = T_{drain} = 64$ time units, which is the same as T_{comp}. In our current design for $N = 8$, we found a better design that requires less T_{load} and T_{drain}.

The designs on the right half of Table 1 are based on optimizing T_c. As a result, they have less total completion time and more #*PE*s than those on the left half of the table. For instance, for $N = 300$, the completion time for the design optimizing T_c requires 7% less completion time and 35% more PEs than the design optimizing T_{comp}. Note that both designs were developed without bounds on #*PE*s.

It is important to point out that the objective used (whether to minimize T_{comp} or to minimize T_c) depends on the application. If the linear processor array is used to evaluate the transitive closure of one matrix, then minimizing T_c will be important. On the other hand, if the processor array is used for pipelined evaluation of transitive closures of multiple matrices, then minimizing T_{comp} may be important. More precisely, we would like to minimize the total completion time for evaluating a sequence of transitive

TABLE 1.

T_{comp}-OPTIMAL AND T_c-OPTIMAL LINEAR ARRAYS FOR FINDING TRANSITIVE CLOSURE OF AN $N \times N$ MATRIX

	GPM: T_{comp}-Optimal Linear-Array Designs					GPM: T_c-Optimal Linear-Array Designs				
			Min T_{comp} Designs		SS10/30 CPU			Min T_c Designs		SS10/30 CPU
N	Periods (t_1, t_2, t_3)	Distances $(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$(T_{load}, T_{comp}, T_{drain})$	#PEs	sec.	Periods (t_1, t_2, t_3)	Distances $(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	$(T_{load}, T_{comp}, T_{drain})$	#PEs	sec.
3	(1, 1, 2)	(0, −1, 1)	(5, 13,5)	3	—	(1, 2, 1)	(0, −1, 1)	(3, 15, 3)	3	—
4	(1, 1, 3)	(0, −1, 1)	(10, 22, 10)	4	—	(1, 3, 1)	(0, −1, 1)	(4, 28, 4)	4	—
8	(1, 1, 5)	(0, −1, 3)	(13, 64, 13)	22	—	(1, 1, 5)	(0, −1, 3)	(15, 64, 15)	22	—
16	(1, 2, 5)	(0, −2, 3)	(51, 166, 51)	46	—	(1, 2, 6)	(0, −1, 5)	(31, 181, 31)	76	—
32	(1, 3, 6)	(0, −3, 5)	(113, 435, 113)	156	—	(1, 2, 9)	(0, −2, 7)	(94, 466, 94)	218	—
64	(1, 5, 7)	(0, −5, 6)	(369, 1198, 369)	379	—	(1, 3, 12)	(0, −3, 11)	(253, 1261, 253)	694	2
100	(1, 5, 11)	(0, −5, 9)	(606, 2278, 606)	892	1	(1, 4, 13)	(1, −4, 12)	(496, 2278, 496)	1387	7
200	(1, 8, 13)	(1, −8, 12)	(1743, 6170, 1743)	2787	7	(1, 5, 21)	(0, −5, 19)	(1195, 6568, 1195)	3782	51
300	(1, 9, 18)	(0, −9, 17)	(2851, 11363, 2851)	5084	26	(1, 7, 22)	(1, −7, 21)	(2393, 11363, 2393)	6878	157

TABLE 2.

#PE-OPTIMAL LINEAR ARRAYS FOR FINDING THE TRANSITIVE CLOSURE OF AN $N \times N$ MATRIX (parameters for *#PE*-optimal designs derived by GPM are shown in Theorem 6.2)

	Lee and Kedem (1990) Designs		Shang and Fortes (1992) Designs		Designs by GPM	
N	$(T_{load}, T_{comp}, T_{drain})$	#PEs	$(T_{load}, T_{comp}, T_{drain})$	#PEs	$(T_{load}, T_{comp}, T_{drain})$	#PEs
3	(5, 17, 5)	5	(3, 11, 3)	3	(5, 13, 5)	3
4	(13, 31, 13)	7	(7, 19, 7)	4	(10, 22, 10)	4
8	(85, 127, 85)	15	(43, 71, 43)	8	(50, 78, 50)	8
16	(421, 511, 421)	31	(211, 271, 211)	16	(226, 286, 226)	16
32	(1861, 2047, 1861)	63	(931, 1055, 931)	32	(962, 1086, 962)	32
64	(7813, 8191, 7813)	127	(3907, 4159, 3907)	64	(3970, 4222, 3970)	64
100	(19405, 19999, 19405)	199	(9703, 10099, 9703)	100	(9802, 10198, 9802)	100
200	(78805, 79999, 78805)	399	(39403, 40199, 39403)	200	(39602, 40398, 39602)	200
300	(178205, 179999, 178205)	599	(89103, 90299, 89103)	300	(89402, 90598, 89402)	300

closures, which includes the total computation time of the set of matrices and the times overlapped between draining the results of the previous matrix and loading the inputs of the next matrix. Similar results on computing a sequence of matrix products can be found in Ganapathy and Wah (1993).

If the objective is to minimize *#PE* in the linear array, then Theorem 6.2, which we state without proof (Ganapathy and Wah, 1992a,b), characterizes the *#PE*-optimal design.

Theorem 6.2. *The parameters*

$$(t_1, t_2, t_3) = (1, 1, N-1) \quad \text{and} \quad (\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, \pm 1, \mp 1)$$

or $(\pm 1, 0, \mp 1)$ *result in a linear array with a primary objective of minimizing the number of PEs, and a secondary objective of minimizing the computation time.*

Table 2 shows the *#PE*-optimal designs obtained by GPM as well as those obtained by Lee and Kedem (LK) (1990) and Shang and Fortes (SF) (1992). In this table, we show the load and drain times, computation time, and *#PE*s for designs derived by the three methods. Π (the schedule vector), S (the PE allocation matrix), and the corresponding parameters in GPM are summarized as follows:

Method	Π	S	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$
LK	$[2N-1, 2, 1]^T$	$[0, 1, 1]^T$	$(1, 2, 2N-4)$	$(1, 1, -2)$
SF	$[N, 1, 1]^T$	$[0, 0, -1]^T$	$(1, 1, N-2)$	$(-1, 0, 1)$
GPM	$[N+1, 1, 1]^T$	$[0, 0, -1]^T$	$(1, 1, N-1)$	$(-1, 0, 1)$

The GPM parameters are computed based on (5.2) and (5.4) assuming the dependencies are $d_1 = [0, 0, 1]^T$, $d_2 = [0, 1, 0]^T$, and $d_3 = [1, -1, -1]^T$.

Table 2 shows that both the SF and GPM designs require the minimum number of PEs. The SF designs, however, were developed based on a different set of parameters. According to Lemma 6.1, the SF designs have a computation time $T_{comp} = (N - 1)(N + 2) + 1$. This computation time is lower than that of the GPM designs characterized by Theorem 6.2. This difference is attributable to the fact that Shang and Fortes assumed that conflict must be avoided only after a variable is first used and before its last use or generation. This is a valid assumption for systems with fast I/O (or where each PE has its own I/O) or in cases where inputs are preloaded and outputs need not be drained or are postdrained. In GPM, we consider both conflicts in computation as well as in the data links. Excluding designs that result in computational and data-link conflicts results in designs that require slightly longer load, drain, and computation times. To illustrate this point, consider the case for $N = 3$ and $N = 4$. The periods and velocities used in the SF design (Shang and Fortes, 1992) lead to spacings $\vec{s}_{3,1} = (N - 1)/(N - 2)$ and $\vec{s}_{3,2} = 1/(N - 2)$. These values of spacings result in data-input conflicts between the tokens $(C_{1,j}, C_{N-1,j-1})$ and $(C_{2,j}, C_{N,j-1})$, $j = 2, 3, \ldots, N$, of the input matrix C (Theorem 6.1).

The space-time diagrams of two linear arrays, one optimizing T_{comp} and the other optimizing T_c, for $N = 3$ are shown in Figs. 4 and 5, respectively. The design in Fig. 4 optimizes T_{comp} and has the parameters $(t_1, t_2, t_3) = (1, 1, 2)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, 1, -1)$. This design minimizes both T_{comp} and $\#PE$, and, therefore, minimizes any objective of the form $\#PE^m \cdot T_{comp}^n$ for $m, n \geq 1$. Note that the load and drain times ($T_{load} = T_{drain} = 5$) are not shown in the diagrams. Further, note that for correct execution of the Floyd–Warshall algorithm, control signals are needed to govern the index-dependent assignments performed by the PEs in the array. These index-dependent assignments are given in Tables I and II in Lee and Kedem (1988).

In Fig. 5, we show a new design that optimizes T_c. This design uses less load and drain times (three units each), but the computation time T_{comp} is higher than in Fig. 4.

Comparing the results shown in Tables 1 and 2, we found, for instance, that for a problem of size 200, the T_c-optimal design is 13.35 times faster than the $\#PE$-optimal design in terms of completion time, and uses 18.9 times more PEs than the $\#PE$-optimal design. (The T_c-optimal design for $N = 200$ requires 8,958 time units and 3,782 PEs, whereas the $\#PE$-optimal design requires 119,602 time units and 200 PEs.) It would be beneficial from a design point of view to develop designs with values of $\#PE$ and T_c in between these extreme values. This is important in practical situations

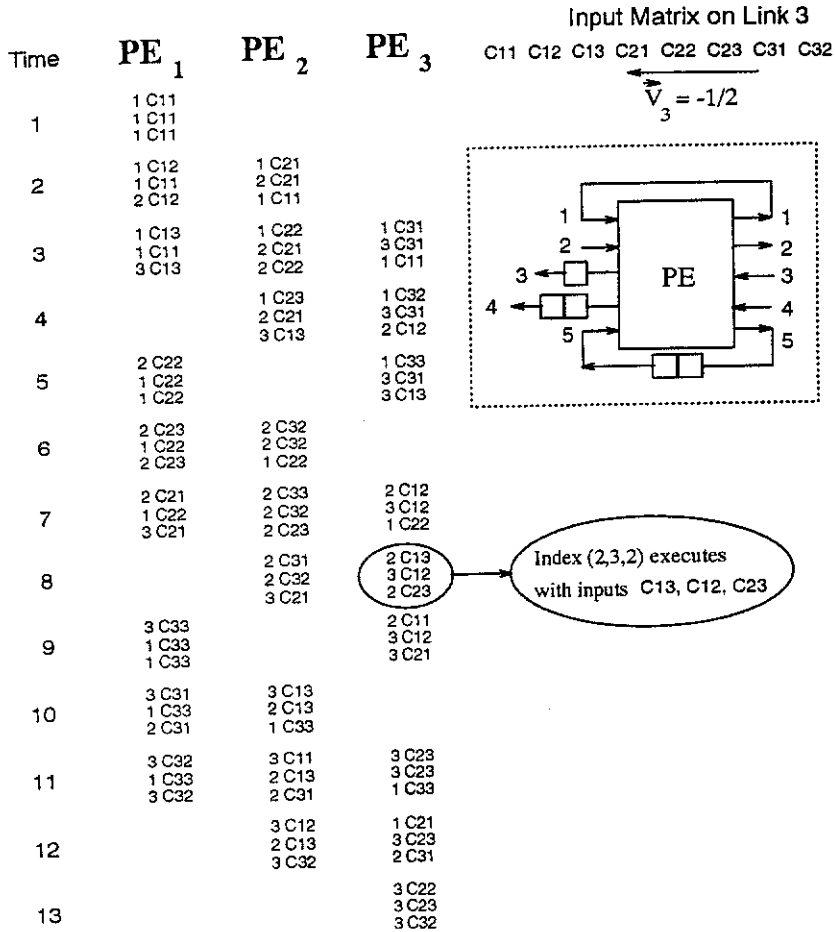

FIG. 4. Linear array to find the transitive closure of a 3×3 matrix. The array is optimal with respect to minimizing computation time, $\#PE$, or $\#PE^m \times T^n$, $m, n \geq 1$.

because a designer might be unwilling to settle for either the large number of PEs required in the minimum-time design or the long completion time of the minimum-processor design. In realistic design situations there may be bounds on the number of processors or the completion time or both. Hence, one possible objective is to have as few processors as possible, so long as the time is within a preset upper limit, T_c^{up} (or T_{comp}^{ub}), and another is to minimize T_c (or T_{comp}) with $\#PE$ less than a given upper bound $\#PE^{up}$.

In the following discussion, let $T_{comp}^{\min}$ and $\#PE^{\max}$ be, respectively, the computation time and $\#PE$ of the minimum-T_{comp} design. Designs with

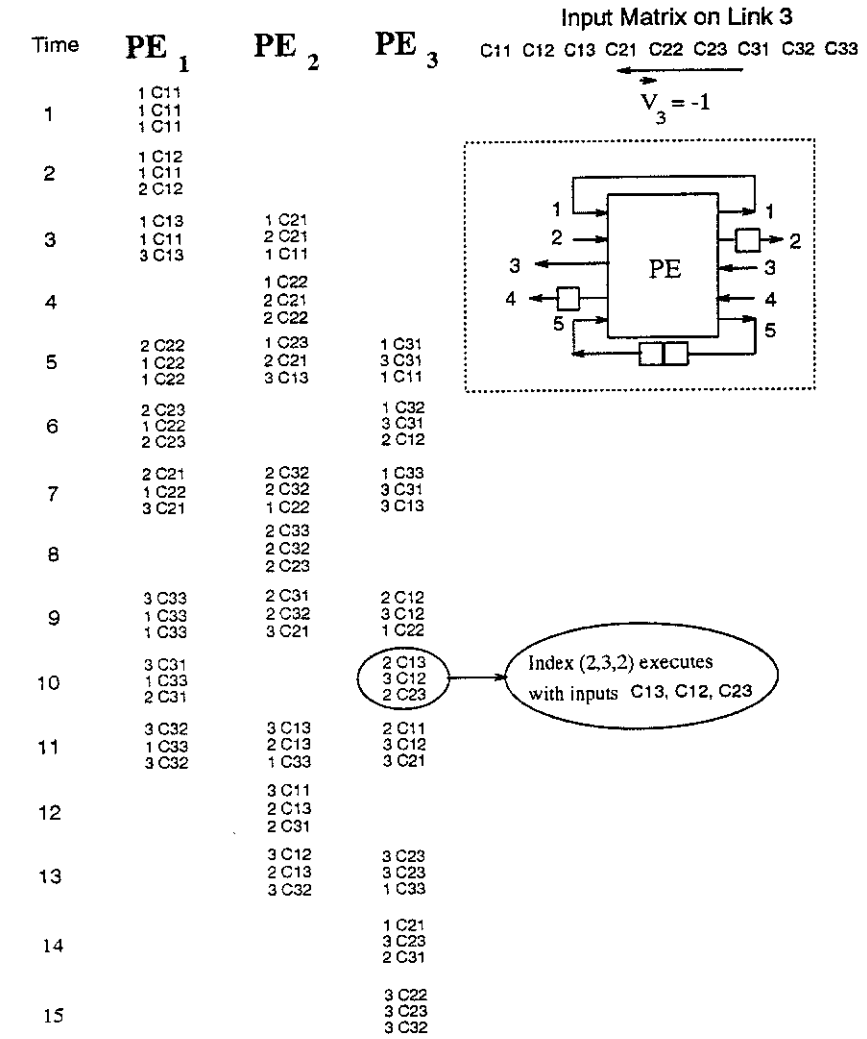

FIG. 5. Linear array to find the transitive closure of a 3×3 matrix. The array is optimal for completion time, which includes load time, computation time, and drain time.

$\#PE > \#PE^{\max}$ would not be useful as their computation times have to be at least $T_{comp}^{\min}$. Let $T_{comp}^{\max}$ and $\#PE^{\min}$ be, respectively, the computation time and the $\#PE$ of the minimum-processor design (from Theorem 6.2 and Lemma 6.2, $\#PE^{\min} = N$). Again, there is no benefit in obtaining designs with $T_{comp} > T_{comp}^{\max}$ as the number of PEs cannot be reduced below $\#PE^{\min}$.

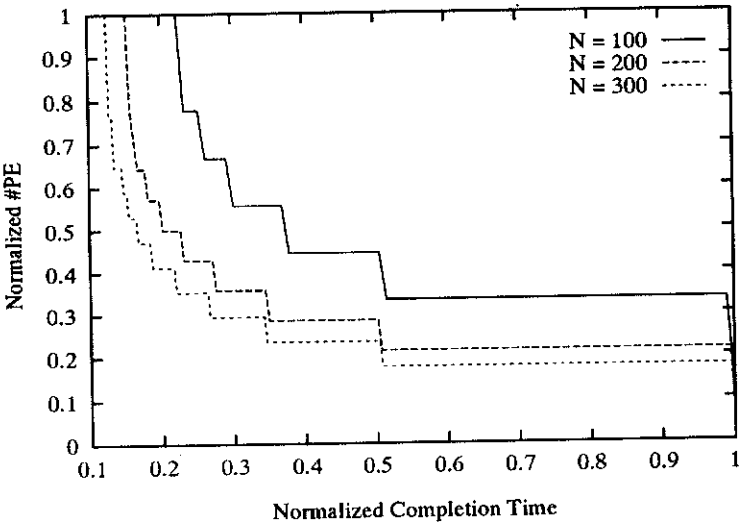

FIG. 6. Performance tradeoffs: Variation in $\#PE$ with time bound T_{comp}^{ub} and variation in T_{comp} with processor bound $\#PE^{ub}$. The plots are given for the three problem sizes $N = 100$, 200, and 300.

We are interested in finding designs with computation time greater than $T_{comp}^{\min}$ and $\#PE$ less than $\#PE^{\max}$.

Figure 6 shows how $\#PE$ varies with T_{comp} for three different problem sizes: $N = 100$, 200, and 300. The y-axis $\#PE$ is normalized by $\#PE^{\max}$, and the x-axis T_{comp} is scaled by $T_{comp}^{\max}$. This lets us compare the different problem sizes uniformly on the same scale. The stepped curves are obtained by bounding T_{comp} and finding the $\#PE$-optimal designs for specific recurrence sizes. There the curves are stepped because there exist only a small and finite number of systolic array configurations that can satisfy the given time constraints. If the goal is to find the $\#PE$-optimal designs, we will have a small number of array configurations; for each configuration, we select the one with the minimum computation time.

Given the bound T_{comp}^{up} (respectively, $\#PE^{up}$) the designer can use Fig. 6 to read off the minimum $\#PE$ (respectively, T_{comp}) required and decide (possibly from a cost perspective) if it is acceptable. Again, the designer could exploit the initial steep decline in the plots to choose an alternative design that trades performance for cost. For instance, the minimum $\#PE$ for $N = 200$ drops by 43% for only a 19% increase in computation time.

If both T_{comp} and $\#PE$ are bounded from above, the design with minimum $\#PE$ for the given time bound is determined using Fig. 6. First, a horizontal line is drawn across the graph for the desired bound on $\#PE$.

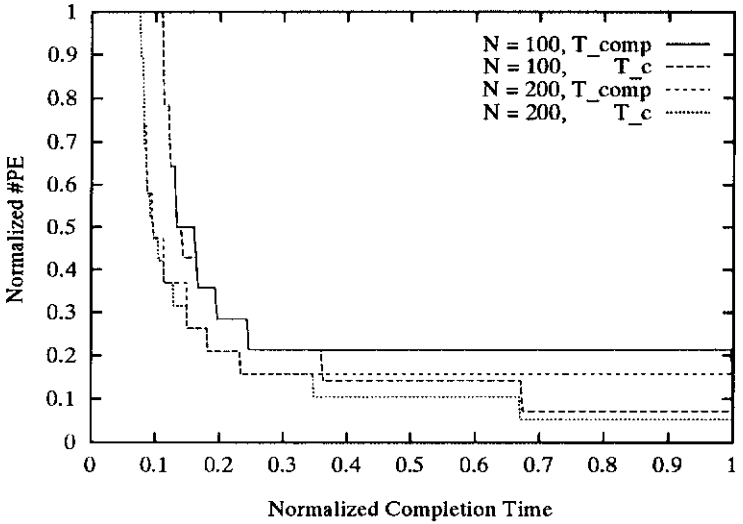

FIG. 7. Different tradeoffs obtained using T_{comp} or T_c as the measure of performance. The plots are given for two problem sizes $N = 100$ and 200.

The intersection between this line and the stepped curve represents the minimum T_{comp} needed for any feasible design. If this minimum T_{comp} is less than the desired T_{comp}, a feasible design can be obtained by the procedure discussed in Section 5. This now represents the best design under both time and processor constraints.

Another observation from Fig. 6 is that the plots for larger N decrease more rapidly than those for smaller N. Hence, for larger values of N, there is a substantial reduction in $\#PE$ (respectively, T_{comp}) for a relatively small increase of the computation time (respectively, $\#PE$) from the optimum. Therefore, for large N, there are more attractive alternatives than the time-optimal or $\#PE$-optimal designs.

Figure 7 shows a similar plot as in Fig. 6 except that here the difference between tradeoffs obtained on T_c and $\#PE$ versus tradeoffs obtained on T_{comp} and $\#PE$ is depicted. Two sets of curves are shown, one for designs that minimize T_{comp} and the other for designs that minimize T_c, for $N = 100$ and $N = 200$, respectively. The y-axis of these curves is normalized with respect to $\#PE$ when T_c is minimum (since these designs require more PEs and less T_c), and the x-axis is normalized with respect to T_c when $T_{comp} = T_{comp}^{\max}$. The graphs show the difference between designs obtained by different objectives. Given a bound T_c^{up}, we can see that the number of processors obtained by minimizing T_c is less than or equal to the number of processors obtained by minimizing T_{comp}.

7. Conclusions

Algorithm-specific parallel processing with linear processor arrays can be systematically achieved with the help of the techniques discussed in this paper. In particular, they are ideally suited to the algorithms that were described as affine recurrences or loop nests in Section 1. They can be conveniently modeled in terms of ordered multidimensional integer sets and matrix algebra that supports the efficient representation and solution of scheduling and processor allocation problems. Of particular importance are the problems of avoiding computational conflicts in processing elements and data communication conflicts in links.

Sections 2 through 4 discuss the dependency method (DM), which is based on linear mappings. We provide conditions that guarantee their correctness, including the absence of computational conflicts. We present closed-form expressions for these conditions which can be used with optimization techniques that use linear integer programming or intelligent searches. Both optimization approaches were discussed along with examples.

In Sections 5 and 6, we describe a general parameter-based approach (GPM) for mapping algorithms with uniform indexing functions to systolic processor arrays. In this method, the behavior of the target array is captured by a set of parameters, and the design problem is formulated as an optimization problem with an objective and a set of constraints specified in terms of the parameters. We show that the parameters in GPM can be expressed in terms of the processor allocation matrix S and the time schedule vector Π in DM, thereby establishing the equivalence between the two representation methods. We present an efficient search procedure for finding T_c-optimal or T_{comp}-optimal (respectively, $\#PE$-optimal) designs for specified bounds on $\#PE$ (respectively, T_c or T_{comp}), as well as optimal designs with a certain monotonicity property on the objective function. The distinct features of GPM lie in its ability to systematically search for optimal designs with specific design requirements of T_c (or T_{comp}) and $\#PE$, and its ability to include constraints on data-link and computational conflicts in the optimization procedure. A similar search procedure that finds T_{comp}-optimal designs has been developed for dependency-based methods (O'Keefe *et al.*, 1991). We believe that a general search procedure that allows tradeoffs between $\#PE$ and T_c in dependency-based methods can be developed for synthesizing uniform recurrences.

In conclusion, we show in this paper two representation methods for synthesizing recurrent computations for linear processor arrays. These two methods differ in their representation power and search procedures. The dependency-based method is more general in its representation power and can be applied to find feasible designs for general (uniform as well

TABLE 3.

COMPARISON BETWEEN DEPENDENCY-BASED VERSUS PARAMETER-BASED METHODS

Feature	Dependency-Based Method Presented in Sections 2–4	Generalized Parameter Method Presented in Sections 5–6
Applicable recurrences	General and applicable to uniform as well as nonuniform recurrences.	Homogeneous uniform recurrences or uniformized affine recurrences.
Representation	Schedule Vector and Allocation Matrix: represented in Cartesian coordinate system with unit vectors as basis vectors; for the dimension-reduction technique discussed in Sections 2–4, the mappings are rank-deficient (i.e., Π and S yield T where $rank(T) > n$).	Periods and Displacements: represented in possibly non-orthogonal coordinate system with dependence vectors as basis vectors; hence, for uniform recurrences, the two representations are equivalent and derivable from each other by a coordinate (linear) transformation.
Characteristics of controls in processor array	Nonuniform in the general case by specifying a general processor allocation matrix; processor arrays derived may, in the general case, have arbitrary speed/direction changes for data tokens and have aperiodic computations.	Uniform controls throughout the processor array, resulting in constant velocities and periodic computations.
Design objective and constraints	Compute-time optimal designs or processor-optimal designs with linear objective function and linear constraints.	General nonlinear objective function and constraints with certain monotonicity properties on the objective function; new constraints have been developed that capture data-link conflicts.
Search methods for finding processor array designs	Choose processor-allocation matrix heuristically, and find schedule vector satisfying processor-allocation constraints; methods of finding designs are based on linear/integer programming or intelligent searches.	Search method is systematic enumeration and pruning on a search space polynomial in complexity with respect to problem size.
Designs obtained	Designs found are optimal in terms of computation time with respect to a given choice of processor-allocation matrix; possible allocation matrices chosen are those that minimize the number of processing elements.	Tradeoffs between processor and computation time (or completion time, including load and drain times) for a specific problem instance can be obtained.
Summary	The two methods presented in this article are equivalent approaches for mapping uniform recurrences. The formulation of the design optimization problem and the search techniques developed are equally applicable in both representations.	

as nonuniform) recurrences. Due to its generality in representation, the search space for finding optimal designs is extremely large. Hence, dependency-based methods find feasible designs heuristically by first specifying how the data tokens should relate to each other. In contrast, the general parameter method is restricted to synthesizing uniform recurrences and affine recurrences that can be uniformized. For this class of recurrences, we can exploit uniformity in data traversal in the processor array. We present an efficient search procedure for finding optimal designs with user-specified requirements on the completion time and on the number of processing elements. Table 3 summarizes the unique features of the two methods.

ACKNOWLEDGMENTS

Research of J. Fortes and W. Shang was supported by Louisiana Education Quality Support Fund LEQSF(1991-93)-RD-A-42, National Science Foundation Grants DC1-8419745 and MIP-9110940, and Innovative Science and Technology, Office of the Strategic Defense Initiative Organization, administered through the Office of Naval Research under contracts 00014-85-K-0588, 00014-88-K-0723 and 00014-90-J-1483. Research of B. Wah and K. Ganapathy was supported by Joint Services Electronics Program contract JSEP N00014-90-J-1270, National Science Foundation grant NSF MIP 92-18715, and an IBM graduate fellowship.

References

Almasi, G. S. and Gottlieb, A. (1989). *Highly Parallel Computing*. Benjamin/Cummings Publishing Company, Inc., Redwood City, CA.

Annaratone, M., Arnould, E., Gross, T., Kung, H. T., Lam, M. S., Menzilcioglu, O., and Webb, J. A. (1987). "The Warp Machine: Architecture, Implementation and Performance," *IEEE Trans. Computers*, C-36, **12**, 1523–38.

Banerjee, U. (1988). *Dependence Analysis for Supercomputing*. Kluwer Academic Publisher, Boston.

Baxter, B., Cox, G., Gross, T., Kung, H. T., O'Hallaron, D., Peterson, C., Webb, J., and Wiley, P. (1990). "Building Blocks for a New Generation of Application-Specific Computing Systems." *Proc. Int'l Conf. on Application Specific Array Processors* (ASAP), pp. 190–201.

Chen, Z. and Shang, W. (1992). "On Uniformization of Affine Dependence Algorithms." *Proc. IEEE Fourth Symposium on Parallel and Distributed Processing*, Arlington, TX, Dec. 1992, pp. 128–37.

Fortes, J. A. B., Lee, E., and Meng, T. (1992). *Proc. of 1992 Application Specific Array Processors*, IEEE Computer Society Press, Los Alamitos, California.

Fortes, J. A. B. and Wah, B. W. (1987). "Systolic Array—From Concept to Implementation." *IEEE Computer*, July 1987, pp. 12–17.

Ganapathy, K. and Wah, B. W. (1992a). "Optimal Design of Processor Arrays for Uniform Recurrences." *Proc. Int'l Conf. on Application-Specific Array Processors*. IEEE Computer Society, Aug. 1992, pp. 636–48.

Ganapathy, K. and Wah, B. W. (1992b). "Synthesizing Optimal Lower Dimensional Processor Arrays." *Proc. Int'l Conf. on Parallel Processing*. CRC Press, Aug. 1992, Vol. 3, pp. 96–103.

Ganapathy, K. and Wah, B. W. (1993). "Designing a Coprocessor for Recurrent Computations." *Proc. Fifth IEEE Symposium on Parallel and Distributed Processing* (in press).

Guibas, L. J., Kung, H. T., and Thompson, C. D. (1979). "Direct VLSI Implementation of Combinatorial Algorithms." *Proc. Caltech Conf. on VLSI*. Caltech, Pasadena, CA, pp. 509–525.

Kannan, R. and Bachem, A. (1979). "Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix." *SIAM J. Computing* **8**(4), 499–507.

Kung, H. T. and Lam, M. (1984). "Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays." *J. Parallel and Distributed Computing* **1**(1), 32–63.

Kung, S. Y., Lo, S. C., and Lewis, P. S. (1987). "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems." *IEEE Trans. Computer* **C-36**, 603–14.

Lee, P. and Kedem, Z. M. (1988). "Synthesizing Linear Array Algorithms from Nested For Loop Algorithms." *IEEE Trans. Computers* **37**(12), 1578–98.

Lee, P. and Kedem, Z. M. (1990). "Mapping nested Loop Algorithms into Multidimensional Systolic Arrays." *IEEE Trans. Parallel and Distributed Systems* **1**(1), 64–76.

Li, G.-J. and Wah, B. W. (1985). "The Design of Optimal Systolic Arrays." *IEEE Trans. Computers* **C-34**, 66–77.

Menzilcioglu, O., Kung, H. T., and Song, S. W. (1989). "Comprehensive Evaluation of a Two-Dimensional Configurable Array." *Proc. 19th Int'l Symposium on Fault-tolerant Computing*, pp. 93–100.

O'Keefe, M. T. and Fortes, J. A. B. (1986). "A Comparative Study of Two Systematic Design Methodologies for Systolic Arrays." *Proc. 1986 Int'l Conf. on Parallel Processing*, pp. 672–5.

O'Keefe, M. T., Fortes, J. A. B., and Wah, B. W. (1991). "On the Relationship Between Systolic Array Design Methodologies." *IEEE Trans. Computers* **41**(12), 1589–93.

Quinton, P. (1989). "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations." *Proc. 11th Annual Symposium on Computer Architecture*, pp. 208–14.

Quinton, P. and Van Dongen, V. (1989). "The Mapping of Linear Recurrence Equations on Regular Arrays." *J. VLSI Signal Processing* **1**(2), 95–115.

Rote, G. (1985). "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths, Matrix Inversion)." *Computing* **34**, 192–219.

Schrijver, A. (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons, New York.

Shang, W. and Fortes, J. A. B. (1992). "On Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays." *IEEE Trans. Parallel and Distributed Systems* **3**(3), 350–63.

Strang, G. (1980). *Linear Algebra and its Applications*, 2nd ed. Academic Press, Boston.

Tucker, L. W. and Robertson, G. G. (1988). "Architecture and Applications of the Connection Machine." *IEEE Computer*, Aug. 1988, pp. 26–38.

Tzen, T. and Ni, L. (1992). "Data Dependence Analysis and Uniformization for Doubly Nested Loops." *Proc. Int'l Conf. on Parallel Processing*. St. Charles, Illinois, pp. 91–99(II).

Valero, M., Kung, S. Y., Lang, T., and Fortes, J. A. B. (1991). *Proc. of 1991 Application Specific Array Processors*. IEEE Computer Society Press, Los Alamitos, California.

Valero-Garcia, M., Navarro, J. J., Llaberia, J. M., and Valero, M. (1989). "Systematic Hardware Adaptation of Systolic Algorithms." *Proc. Int'l Symposium on Computer Architecture*. ACM/IEEE, pp. 96–104.

Wong, Y. and Delosme, J.-M. (1992). "Transformation of Broadcasts into Propagations in Systolic Arrays." *J. Parallel and Distributed Computing* **14**(2), 121–45.

Xing, Z. and Shang, W. (1993). "An Algorithm for Accurate Data Dependence Test." *Proc. IEEE Int'l Conf. on Application Specific Array Processors*, Oct. 1993, Italy (in press).

Xue, J. (1993). "A New Formulation of the Mapping Conditions for the Synthesis of Linear Systolic Arrays." *Proc. IEEE Int'l Conf. on Application-Specific Array Processors*, Oct. 1993, Italy (in press).

Yaacoby, Y. and Cappello, P. R. (1988). "Scheduling a System of Affine Recurrence Equations onto a Systolic Array." *Proc. Int'l Conf. on Systolic Arrays*, San Diego, CA, May 1988, pp. 373–82.

Yang, Z., Shang, W., and Fortes, J. A. B. (1992). "One-to-One Time Mappings of Nested Algorithms into Lower Dimensional Processor Arrays." *Proc. of the Sixth IEEE Int'l Parallel Processing Symposium*, March 1992, Beverly Hills, CA, pp. 156–64.