Optimal Synthesis of Algorithm-Specific Lower-Dimensional Processor Arrays

Kumar N. Ganapathy and Benjamin W. Wah, *Fellow, IEEE*

Abstract—Processor arrays are frequently used to deliver high performance in many applications with computationally intensive operations. This paper presents the *General Parameter Method* (*GPM*), a systematic parameter-based approach for synthesizing such algorithm-specific architectures. GPM can synthesize *processor arrays of any lower dimension* from a *uniform-recurrence description* of the algorithm. The design objective is a general *nonlinear and nonmonotonic user-specified function*, and depends on attributes such as computation time of the recurrence on the processor array, completion time, load time, and drain time. In addition, bounds on some or all of these attributes can be specified. GPM performs an efficient search of *polynomial complexity* to find the *optimal* design satisfying the user-specified design constraints. As an illustration, we show how GPM can be used to find optimal linear processor arrays for computing transitive closures. We consider design objectives that minimize computation time, or processor count, or completion time (including load and drain times), and user-specified constraints on number of processing elements and/or computation/completion times. We show that GPM can be used to obtain optimal designs that trade between number of processing elements and completion time, thereby allowing the designer to choose a design that best meets the specified design objectives. We also show the equivalence between the model assumed in GPM and that in the popular dependence-based methods [1], [2]. Consequently, GPM can be used to find optimal designs for both models.

Index Terms—Design constraints, objective function, optimal design, polynomial-time search, processor arrays, transitive closure, uniform recurrence equations.

1 Introduction

PROCESSOR arrays (or systolic arrays), due to their structural regularity and suitability for VLSI implementation, are frequently used to meet performance requirements of many computationally intensive applications. The fundamental concept behind a processor array architecture is that the *Von-Neumann* bottleneck is greatly alleviated by repeated use of a fetched data item in a physically distributed array of processing elements [3]. This paper discusses systematic ways of mapping regular and computationally-intensive algorithms onto specialized processor arrays. An overview of various existing methods to generate processor arrays systematically can be found in reference [4].

The techniques discussed here apply to algorithms described as recurrences, either by mathematical expressions or by high-level-language programs. Section 1.1 provides a precise characterization of the class of algorithms for which our results are valid. 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 processor arrays of arbitrary dimensions. We choose to study linear arrays because they are easier to build and program than arrays of higher dimension.

• K. Ganapathy is with the Telecommunications Division, Rockwell International, 4311 Jamboree Road, P.O. Box C, M/S 501-364, Newport Beach, CA 92658-8902. E-mail: kumar@nb.rockwell.com.
• B.W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801. E-mail: wah@manip.crhc.uiuc.edu.

Manuscript received Jan. 13, 1992; revised Oct. 1, 1994.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number D95074.

The general notation used in this paper is as follows. Vectors are in lower case with arrows on top, and matrices are in upper-case bold font. The transpose of vector $\vec{v}$ and matrix $\mathbf{M}$ are denoted by $\vec{v}^t$ and $\mathbf{M}^t$, respectively. The absolute value of vector $\vec{v}$ is denoted by $|\vec{v}|$, and notation $\vec{v} \geq \vec{u}$ means that every component of $\vec{v}$ is greater than or equal to the corresponding component of $\vec{u}$. Vector $\vec{0}$ denotes a row or column vector whose entries are all zeroes. The dimensions of vector $\vec{0}$, and whether it denotes a row or column vector, are implied by the context in which it is used. The scalar product of two vectors $\vec{v}_1$ and $\vec{v}_2$, and the product of vector $\vec{v}$ and matrix $\mathbf{M}$ are written (without transposes) as $\vec{v}_1 \cdot \vec{v}_2$ and $\vec{v} \cdot \mathbf{M}$ (or $\mathbf{M} \cdot \vec{v}$), respectively. The product of two matrices $\mathbf{M}_1$, $\mathbf{M}_2$, and a scalar s and a vector $\vec{v}$ are simply written as $\mathbf{M}_1$, $\mathbf{M}_2$, and $s\vec{v}$ without any dot symbol.

1.1 Algorithm Model

Affine dependence algorithms can be used to model a large number of computation-intensive applications in image processing, digital signal processing, and other scientific applications. Such algorithms can be described as nested DO loops as follows.

```
DO ($j_1 = l_1, u_1$ ; $j_2 = l_2, u_2$ ; $\cdots$ ; $j_n = l_n, u_n$)
    $H_1(\vec{J})$; $H_2(\vec{J})$; ...; $H_t(\vec{J})$;
END
```

The column vector $\vec{J} = [j_1, j_2, \cdots, j_n]^t$ is the index vector (or index point). $H_i(\vec{J})$, $i = 1, \cdots, t$, are t assignment statements in iteration $\vec{J}$ having the form

$$Z_i\left(y(\vec{J})\right) = \phi\left[Z_1\left(x_1(\vec{J})\right), \ldots, Z_r\left(x_r(\vec{J})\right)\right], \quad 1 \leq i \leq r. \quad (1)$$

Affine recurrence equations (*ARE*) with a convex polyhedral domain can be used to model the above program if

1) all loop bounds l_i and u_i are affine functions of loop variables $j_1, \ldots, j_{i-1}$;
2) indexing functions $y()$ and $x_k()$, $k = 1, \cdots, r$, are affine functions of the form $\mathbf{A} \cdot \vec{J} + \vec{d}_i$; and
3) branch statements do not go outside the loop containing the branch statement.

If iteration $\vec{J}$ depends on iteration $\vec{J}'$, then this dependence can be described by a dependence vector $\vec{d} = \vec{J} - \vec{J}'$, which is the vector difference of the index vectors of these two iterations. The dependencies in the algorithm can be shown by a dependence graph (DG) over an n-dimensional (nD) domain (integer lattice), where nodes are labeled by index vectors corresponding to the operations in the innermost loop body, and arcs correspond to the loop-carried dependencies between two instances of the loop body. Hence, the loop body for scheduling is the set of statements in loop nests enclosing all the branch statements.

Uniform dependence algorithms or *uniform recurrence equations* (*URE*) form a subclass of AREs, where indexing functions $y()$ and $x_k()$ are of the form $\vec{J} - \vec{d}$ (matrix $\mathbf{A}$ is the identity matrix now), and $\vec{d}$ is a constant vector of n elements. Hence, each of the statements $H_i(\vec{J})$ is given by

$$Z_i(\vec{J}) = \phi\left[Z_1(\vec{J} - \vec{d}_1), \ldots, Z_r(\vec{J} - d_r)\right]. \tag{2}$$

There exist "uniformization" techniques for transforming AREs to UREs. (See, for example, reference [5].) The basic idea is to select a few basic integral vectors (which are the uniform dependencies) such that all affine dependencies of the ARE can be expressed as non-negative integer linear combinations of the basis vectors. This uniformization also removes the undesirable broadcasts of data in a VLSI processor array.

In this paper, we focus on algorithms that can be modeled as uniform recurrences and affine recurrences that can be uniformized. Hence, the starting point of our mapping assumes a convex polyhedral domain and a set of constant dependence vectors collected into a matrix called the *dependence matrix D*.

EXAMPLE 1. Matrix multiplication of two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ is a well known example of an URE, where

$$C(i, j, k) = C(i, j, k-1) + \mathbf{A}(i, k)\mathbf{B}(k, j), \quad 1 \le i, j, k \le N. \tag{3}$$

The index set consists of all the integer points in a cube of side N. Input $\mathbf{A}(i, k)$ (*resp.*, $\mathbf{B}(k, j)$) is used in several computations to generate $C(i, j, k)$ for all values of j (*resp.*, i). After pipelining and localizing the affine dependencies, $[0, j, 0]^t$ and $[i, 0, 0]^t$, we get

$$\mathcal{A}(i, j, k) = \mathcal{A}(i, j-1, k); \quad \mathcal{B}(i, j, k) = \mathcal{B}(i-1, j, k)$$
$$C(i, j, k) = C(i, j, k-1) + \mathcal{A}(i, j, k)\mathcal{B}(i, j, k), \tag{4}$$

which is a set of uniform recurrence equations. □

EXAMPLE 2. Consider a 3-Dimensional (3D) recurrence with $n = 3, r = 5$.

$$Z(k, i, j) = \mathbf{X}(k, i)\mathbf{Y}(j, k) + Z(k-1, i+1, j+1) + Z(k-1, i+1, j) + Z(k-1, i, j+1) \tag{5}$$

After pipelining and uniformization, (5) becomes

$$Z(k, i, j) = \mathcal{X}(k, i, j-1)\, \mathcal{Y}(k, i-1, j) + Z(k-1, i+1, j+1) + Z(k-1, i+1, j) + Z(k-1, i, j+1) \tag{6}$$

The dependence vectors collected into a matrix are

$$D = \begin{array}{c} \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & 0 \\ 1 & 0 & -1 & 0 & -1 \end{bmatrix} \\ \begin{array}{ccccc} \mathcal{X} & \mathcal{Y} & \mathcal{Z} & \mathcal{Z} & \mathcal{Z} \end{array} \end{array} \tag{7}$$

This example is used as a running example throughout this paper.

1.2 Previous Work

There has been a lot of research in developing design methods to map uniform dependence algorithms to processor arrays. Most of these methods are based on or derived from the *Dependency Method* (*DM*) [1], [2]. In DM, the problem of mapping an algorithm to a processor array is characterized by a *linear mapping matrix* $\mathbf{T} = \begin{bmatrix} \vec{\Pi} \\ \mathbf{S} \end{bmatrix}$ where $\vec{\Pi}$ is the schedule vector and $\mathbf{S}$ is the allocation matrix. The design of the array is then equivalent to determining the elements of $\mathbf{T}$. This general representation of a feasible design as a particular mapping matrix allows DM to be applied to uniform as well as nonuniform recurrences. However, in DM, the generality in representation leads to large search spaces for optimal designs, as the problem of finding optimal designs is posed as an integer programming problem [6], [7]. In contrast, the method presented in this paper, the *General Parameter Method* (*GPM*), is restricted to uniform recurrences, but can be used to generate optimal designs for user-specified objectives (including nonmonotonic and nonlinear ones) using efficient search techniques of polynomial complexity.

There have been several earlier attempts to map algorithms onto lower dimensional arrays [8], [7], [9]. Important steps towards a formal solution were first made by Lee and Kedem [8]. They presented the concept of data-link collisions (two data tokens contending for the same link simultaneously) and conditions to avoid them. They also presented a method that analyzes all computations in the domain of the recurrence in order to detect computational conflicts (two computations scheduled to execute simultaneously in the same processor). To identify feasible designs, they provided necessary and sufficient conditions for designs that avoid computational and data-link conflicts. However, they did not present any systematic procedure for finding optimal designs. Subsequently, Shang and Fortes [6] have developed closed-form conditions for a mapping to be free of computational conflicts. These closed-form conditions also eliminate data-link conflicts for active data[1] participating in the computations.

In general, in DM, feasible designs are found heuristically by first specifying a "good" allocation matrix $\mathbf{S}$, and

1. The lifetime of a data token in a processor array can be viewed as consisting of an active phase, where the token is involved in its chain of computations, and a passive phase, where the token is moving from the input peripheral processor to becoming active, or is moving to an output peripheral processor after its active phase.

then subsequently determining the schedule vector $\vec{\Pi}$ that minimizes the computation time. Note that the number of choices for matrix **S** could be very large or even infinite, making it difficult (or impossible) to enumerate over them.

Initial work on parameter-based methods was done by Li and Wah [10] for a restricted set of uniform recurrences. They considered specifically 3D and 2D recurrences and mapped them to 2D and 1D processor arrays, respectively. The structure of the recurrence was such that the dependence vectors were unit vectors and the dependency matrix, an identity matrix. This paper generalizes the above initial work into a powerful and efficient array-synthesis technique (called General Parameter Method or GPM) by making important and non-trivial extensions (listed in Section 6). In addition, we show the equivalence between DM and GPM by providing necessary transformations between the parameters in GPM to those in DM, and vice versa. DM can be considered as a mapping problem in the Cartesian coordinate system with unit vectors as basis vectors, whereas GPM can be considered as mapping in a possibly non-orthogonal coordinate system with dependence vectors as basis vectors.

The potential simplicity of GPM over DM is explained by observing that in mapping an n-D algorithm to an m-D processor array, the number of variables to be determined in DM is $(m + 1) \times n$, whereas the number of parameters in GPM is $(m + 1) \times g$, where $g = rank(\mathbf{D})$. Since $g \leq n$ (as **D** is an $n \times r$ matrix), the number of variables in GPM is often less than that in DM, and is at worst equal to the number of variables in DM. Hence, there is potential reduction in complexity by performing the transformation, especially if there are only a few dependence vectors in a high-dimensional space.

Our transformation between GPM and DM extends the work of O'Keefe, Fortes, and Wah [11], who showed the equivalence between DM and GPM for 2D and 3D uniform recurrences. Our transformation also allows efficient search strategies developed in GPM to be used to find optimal designs in DM. Consequently, designers familiar with DM can obtain better (or optimal) array designs using GPM.

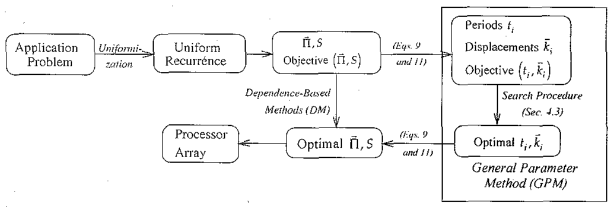


Fig. 1. Application of GPM to find optimal designs in DM.

Referring to Fig. 1, after defining an objective (possibly non-linear and nonmonotonic) in terms of the representation chosen (i.e., $\vec{\Pi}$ and **S**), the designer converts the objective in terms of the parameters of GPM using (9) and (11) (to be discussed in the next section). Once the objective and variables have been converted, GPM is used to generate optimal arrays efficiently. The solutions obtained by GPM are then converted to $\vec{\Pi}$ and **S** in DM using (9) and (11) again. This step involves solving two sets of simultaneous equations for $\vec{\Pi}$ and **S** from the periods and displacements in GPM, and has a worst-case complexity of $O(n^3)$.

The next three sections describe the parameters used in GPM, the constraints that must be satisfied for correct operation, the specification of the objective function, and the search strategy. We assume that processing elements are equally spaced in m dimensions with unit distance between directly connected processing elements, and that buffers between directly connected processing elements, if any, are assumed to be equally spaced along the link.

2 General Parameter Method: Parameters

The intuition behind GPM is as follows. It is known that the semantics of processor arrays can be formally described by uniform recurrence equations; i.e., processor arrays are *"isomorphic"* to uniform recurrences. This implies that as long as the computations defined by an URE are well-formed, there is a direct mapping from the recurrence to a processor array. In fact, this mapping is equivalent to a *linear* transformation of the index set. Hence, for a linear mapping, the time (*resp.*, 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 $\vec{\Pi} \cdot \vec{d}$ (*resp.*, $\mathbf{S} \cdot \vec{d}$) independent of index points $\vec{I}_1$ and $\vec{I}_2$. For recurrences with uniform indexing functions (i.e., UREs and uniformized AREs), the dependences are constant vectors and homogeneous (i.e., the set of dependence vectors at any point in the index set is the same as any other in the index set). Thus, the computation of a recurrence on a processor array is periodic in time and space along dependence vectors in the index space. This periodicity is succinctly captured and exploited in GPM, which considers mapping problems in a possibly nonorthogonal coordinate system with dependence vectors as basis vectors. In other words, in GPM, a representation that captures the above periodicity is used, which allows the optimal target array to be found efficiently.

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

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

$$t_j = \tau_c(\vec{I} + \vec{d}_j) - \tau_c(\vec{I}), \quad j = 1, 2, \cdots, r. \tag{8}$$

The number of periods defined is equal to r, the number of dependencies in the algorithm. In terms of DM, period t_j is related to $\vec{\Pi}$, the schedule vector in DM, by the following equation [12].

$$t_j = \vec{\Pi} \cdot \vec{d}_j. \tag{9}$$

PARAMETER 2: VELOCITY. $\vec{V}_j$, velocity of a datum along dependence vector $\vec{d}_j$, $j = 1, 2, \cdots, r$, is defined as the directional distance passed during a clock cycle. Since PEs are at unit distance from their neighbors, and buffers (if present) must be equally spaced between PEs, the magnitude of the velocity must be a rational number of the form x/y, where x and y are integers and $x \le y$ (to prevent broadcasting). This implies that in y clock cycles, a datum propagates through x PEs and $y - x$ buffers. All tokens of the same variable have the same velocity (both in speed and direction) which is constant during the execution in the processor array. The total number of velocity parameters is r (one for each dependence vector) with each velocity an m-element vector, where m is the dimension of the processor array. Hence, velocity $\vec{V}_j$ is given by,

$$\vec{V}_j = \frac{\vec{k}_j}{t_j}, \quad j = 1, 2, \cdots, r, \tag{10}$$

where $\vec{k}_j$ is the (vector) distance between execution locations of the source and sink index points of $\vec{d}_j$. In the notation of DM, **S**, the allocation matrix, is related to $\vec{k}_j$ and $\vec{d}_j$ as follows.

$$\vec{k}_j = \mathbf{S} \cdot \vec{d}_j. \tag{11}$$

PARAMETER 3: SPACING OR DATA DISTRIBUTION. Consider variable Ω_i pipelined along dependence vector $\vec{d}_i$, $1 \le i \le r$. Data token $\Omega_i(\vec{I} - \vec{d}_i)$ is used at index points $\vec{I} + t\vec{d}_i$, $t = \cdots, -1, 0, 1, \cdots$, in computing the recurrence. In other words, this token moves through the processors that use datum Ω_i at index points $(\vec{I} + t\vec{d}_i)$. Consider another token $\Omega_i(\vec{I} - \vec{d}_j)$ of the same variable Ω_i that is used at index points $(\vec{I} - \vec{d}_j + t\vec{d}_i)$, $j \ne i$. The directional distance in the processor space from token $\Omega_i(\vec{I} - \vec{d}_j)$ to $\Omega_i(\vec{I} - \vec{d}_i)$ is defined as spacing parameter $\vec{S}_{i,j}$. Since there are r variables Ω_i, $1 \le i \le r$, each associated with dependence vector $\vec{d}_i$, there are $r - 1$ nontrivial spacing parameters for each variable and one trivial spacing parameter, $\vec{S}_{i,i} = \vec{0}$. These denote the r distances for variable i: $\Omega_i(\vec{I} - \vec{d}_j) \to \Omega_i(\vec{I} - \vec{d}_i)$, $j = 1, 2, \cdots, r$. Each spacing parameter $\vec{S}_{i,j}$ is an m-D 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. In the notation of DM, we have

$$\begin{aligned}
\vec{S}_{i,j} &= \vec{V}_j t_j - \vec{V}_i t_j && \text{(from Theorem 1 to be presented in Section 3.1)}\\
&= \vec{k}_j - \vec{V}_i t_j && \text{(from (10))}\\
&= \mathbf{S} \cdot \vec{d}_j - \frac{\vec{k}_i}{t_i} t_j && \text{(from (10) and (11))}\\
&= \mathbf{S} \cdot \vec{d}_j - \frac{\vec{\Pi} \cdot \vec{d}_j}{\vec{\Pi} \cdot \vec{d}_i} \mathbf{S} \cdot \vec{d}_i && \text{(from (9) and (10))}
\end{aligned} \tag{12}$$

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

EXAMPLE 3. For the recurrence in (6), the parameters defined are as follows. There are five periods t_1, t_2, t_3, t_4, t_5, and five velocities $\vec{V}_1, \vec{V}_2, \vec{V}_3, \vec{V}_4, \vec{V}_5$. There are 25 spacing parameters $\vec{S}_{i,j}$, $i, j = 1, 2, 3, 4, 5$, where $\vec{S}_{i,i} = \vec{0}$. For instance, for variable $\mathcal{X}$, $\vec{S}_{1,2}, \vec{S}_{1,3}, \vec{S}_{1,4}, \vec{S}_{1,5}$ define distances $(\mathbf{X}(k, i) \to \mathbf{X}(k, i - 1))$, $(\mathbf{X}(k, i) \to \mathbf{X}(k - 1, i + 1))$, $(\mathbf{X}(k, i) \to \mathbf{X}(k - 1, i + 1))$, and $(\mathbf{X}(k, i) \to \mathbf{X}(k - 1, i))$, respectively. □

3 GENERAL PARAMETER METHOD: CONSTRAINT EQUATIONS

In Section 2, a set of $r^2 + r$ parameters have been introduced to define a mapping on the target processor array. Assignment of values to the parameters defines a specific processor array with a particular number of processors, buffers, and data-input pattern. It is also easy to see that all processor 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 performance, and the problem of array design has been reduced to that of choosing appropriate parameter values.

The choice of values for all $(r^2 + r)$ parameters are not independent of each other. In this section, constraint equations relating the parameters are given such that the set of values for the parameters is meaningful and defines a valid processor array. Theorems 1 and 2 provide the fundamental space-time relationship that must be satisfied by the parameters for correct systolic processing. Computational and data-link conflicts are avoided by enforcing the condition in Theorem 3.

The following notation is introduced to simplify the presentation of the theorems. Let $\vec{T} = [t_1, t_2, \cdots, t_r]^t$ be a vector composed of periods, and let $\mathbf{K} = [\vec{k}_1, \vec{k}_2, \ldots, \vec{k}_r]$ be a matrix (of size $m \times r$, where m is the dimension of the processor array) composed of displacements $\vec{k}_i = \vec{V}_i t_i$. Note that $\vec{T}$ is an $r \times 1$ column vector, and that $\vec{k}_i$ is an $m \times 1$ column vector. The displacement $\vec{k}_i$ is synonymous with 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 $\vec{V}_i$.

3.1 Constraints for Correct Systolic Processing of UREs

The following theorem relates the parameters defined in GPM in the necessary conditions for correct systolic processing.

THEOREM 1. *The parameters velocities, spacings, and periods must satisfy the following constraint equations for correct systolic processing of the uniform recurrence equation:*

$$\vec{V}_i t_i = \vec{V}_j t_i + \vec{S}_{j,i}, \quad i, j = 1, 2, \cdots, r. \qquad (13)$$

PROOF. See reference [13]. □

These constraints ensure that in computing an index point $\vec{I}$ at any processor in the array, all the participating data tokens are present at the processor at the same time, moving from their respective processors where they were used earlier. A total of r^2 vector constraints are obtained from Theorem 1.

3.2 Constraints for Linearly Dependent Dependence Vectors

Let $\mathbf{S} = [\vec{S}_{i,j}]$, $i, j = 1, 2, \cdots, r$, 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 by definition that $\vec{S}_{i,i} = 0$. Let $\mathbf{S}_i$ be the ith "row" of $\mathbf{S}$; i.e., $\mathbf{S}_i = [\vec{S}_{i,1}, \vec{S}_{i,2} \cdots \vec{S}_{i,r}]$ (where $\mathbf{S}_i$ 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 1, $\mathbf{S}_i$ can be written in matrix form as

$$\mathbf{S}_i = \mathbf{K} - \vec{V}_i \otimes \vec{T}, \qquad (14)$$

where $\vec{T}$ is a vector composed of periods, and $\otimes$ is the outer product 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 dependence vectors in the recurrence are not linearly independent.

Let g be the rank of dependency matrix $\mathbf{D}$. Therefore, $\mathbf{N}$, the *null space* of $\mathbf{D}$, has $(r - g)$ columns (as $\mathbf{D}$ has r columns). Let $N = [\vec{\alpha}_1, \vec{\alpha}_2 \cdots \vec{\alpha}_{r-g}]$ be an $r \times (r - g)$ matrix, where $\vec{\alpha}_i$, $i = 1, 2, \cdots (r - g)$, are the basis vectors of the null space of $\mathbf{D}$. Hence,

$$\mathbf{D} \cdot \vec{\alpha}_i = 0, \quad 1 \le i \le (r - g). \qquad (15)$$

THEOREM 2. *The periods t_i and the displacements $\vec{k}_i$ are related as follows:*

$$\vec{T} \cdot \mathbf{N} = \vec{0} \qquad (16)$$

$$\mathbf{KN} = \vec{0} \qquad (17)$$

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

PROOF. See reference [13]. □

Theorem 2, therefore, provides a total of $2(r - g)$ constraints: $(r - g)$ scalar constraints and $(r - g)$ vector constraints.

The following corollary shows the constraints on spacings that follow from Theorem 2. In fact, these constraints can be shown to be equivalent to those in Theorem 2. The implication of this corollary is that, of the r spacing parameters for each variable, only $(g - 1)$ of them are independent, one of them is zero, and the rest can be expressed as linear combinations of the $(g - 1)$ independent ones.

COROLLARY 1. *The spacing parameters $S_i = [\vec{S}_{i,1} \cdots \vec{S}_{i,r}]$ are constrained by the equations $\mathbf{S}_i\mathbf{N} = 0$, $i = 1, 2, \cdots, r$, where $\mathbf{N}$ is the matrix consisting of the basis vectors of the null space of $\mathbf{D}$.*

PROOF. From (14), we know that $\mathbf{S}_i = \mathbf{K} - \vec{V}_i \otimes \vec{T}$. Using the property of outer products that $(\vec{a} \otimes \vec{b}) \cdot \vec{c} = (\vec{b} \cdot \vec{c})\vec{a}$, we get

$$\mathbf{S}_i \cdot \vec{\alpha}_i = \mathbf{K} \cdot \vec{\alpha}_i - (\vec{T} \cdot \vec{\alpha}_1)\vec{V}_i = 0$$

for any column $\vec{\alpha}_i$ of matrix $\mathbf{N}$. The corollary is proved by applying Theorem 2. □

EXAMPLE 4. From Theorem 1, the constraint equations for the recurrence in (5) (excluding the trivial constraint $\vec{V}_1 t_1 = \vec{V}_1 t_1 + \vec{S}_{1,1}$) are

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

Similarly, there are 16 additional equations related to $\vec{V}_2 t_2, \vec{V}_3 t_3, \vec{V}_4 t_4$, and $\vec{V}_5 t_5$.

$\mathbf{D}$ defined in (7) has rank 3. Hence, $\mathbf{N}$ comprises of two basis vectors.

$$\mathbf{N} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}$$

From Theorem 2, the additional constraints are

$$t_4 = t_1 + t_3 \qquad t_5 = t_2 + t_3 \qquad (18)$$

$$\vec{k}_4 = \vec{k}_1 + \vec{k}_3 \qquad \vec{k}_5 = \vec{k}_2 + \vec{k}_3 \qquad (19)$$

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 1 and 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 $r^2 + (r - g)$ vector parameter values can be determined from the chosen scalar and vector parameters. Since the performance of the design can naturally be expressed in terms of the periods and displacements, our strategy is to choose the g periods and g displacements to optimize a given performance criterion. The remaining $(r - g)$ periods, $(r - g)$ displacements, and all of the spacings can be determined from Theorems 1 and 2. All the vector equations are solved in m-D space in order to obtain m-D vector parameters.

3.3 Constraints to Govern Valid Space-Time Mappings

The validity of a 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 it depends on have been executed. In DM, $\vec{\Pi} \cdot \mathbf{D} > 0$.
2) **Avoidance of Computational Conflicts.** No two index

points should be executed at the same processor at the same time. In DM, $\vec{\Pi}\cdot\vec{I}_1 = \vec{\Pi}\cdot\vec{I}_2$, implying that $\mathbf{S}\cdot\vec{I}_1 \neq \mathbf{S}\cdot\vec{I}_2$.

3) **Avoidance of Data-Link Conflicts.** No two data tokens should contend for a given link at the same time.

Having established the parameters and the basic relationship among them in Theorems 1 and 2, we show how the fundamental conditions for valid space-time mappings are satisfied in GPM.

By definition, periods denote the time difference between the source and sink of dependencies. Hence, the precedence constraint is satisfied by simply enforcing $t_i \geq 1$, $i = 1,\cdots, r$. 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 are in the same physical location and 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 non-stationary variable, then there are data-input conflicts for the tokens of that variable. Otherwise, all variables are stationary, and the entire computation is executed in one processor; i.e., there is no processor array. Hence, by enforcing that no data-input conflicts exist, both computational and data-link conflicts are avoided. Theorem 3 below presents conditions under which data-input conflicts can be eliminated.

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

$$\mathbf{S}_i' = \left[\vec{S}_{i,1}\vec{S}_{i,2}\cdots\vec{S}_{i,g-1}\right] \tag{20}$$

where $\vec{S}_{i,1}\vec{S}_{i,2},\ldots,\vec{S}_{i,g-1}$ are the $(g-1)$ independent spacings. Let $\vec{\alpha},\vec{\beta},\vec{\gamma}$ be vectors with $(g-1)$ integral elements. Let L_j, U_j, $j = 1, 2, \cdots, g-1$, be defined such that the position of all the tokens of the input matrix can be represented by $\sum_{j=1}^{g-1}\vec{S}_{i,j}\beta_j$, where $L_j \leq \beta_j \leq U_j$, and L_j and U_j are functions of the size of the input matrix.

THEOREM 3. *Data-input conflicts occur in the input matrix of nonstationary input i if and only if* $\mathbf{S}_i'\cdot\vec{\alpha} = \vec{0}$, *where* $\vec{\alpha} = [\alpha_1,\alpha_2,\ldots,\alpha_{g-1}]^t \neq \vec{0}$, *and* $\alpha_i \in [(L_i - U_i), \ldots, (L_i + U_i)]$ *for all i such that* $1 \leq i \leq g-1$ [14].

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

$$\begin{aligned}\text{Data-input conflicts} &\Leftrightarrow S_i'\cdot\vec{\beta} = S_i'\cdot\vec{\gamma},\\ &\quad\text{where } \vec{\beta}\neq\vec{\gamma} \text{ and } L_i \leq \gamma_i,\ \beta_i \leq U_i\\ &\Leftrightarrow S_i'\cdot(\vec{\beta}-\vec{\gamma}) = \vec{0}\\ &\Leftrightarrow S_i'\cdot\vec{\alpha} = \vec{0},\\ &\quad\text{where } \vec{\alpha} = \vec{\beta}-\vec{\gamma}\neq\vec{0},\ \alpha_i \in [(L_i - U_i),\ldots,(L_i + U_i)].\end{aligned}$$

Note that in Theorem 3, we have defined conservative bounds on α_i. Better estimates can be obtained [15] and will result in less overhead when the conditions in Theorem 3 are checked in the design process.

EXAMPLE 5. For the recurrence in (5), if the array sought is 1D, then the spacing parameters are all 1D scalars. Let $\vec{S}_{1,2}$ and $\vec{S}_{1,5}$ be the two independent spacings for input **X**, and we choose the values of $L_1 = L_2 = 1$ and $U_1 = U_2 = N$. According to Theorem 3, data-input conflicts occur in input **X** if and only if

$$\left[\vec{S}_{1,2}\vec{S}_{1,5}\right]\begin{bmatrix}\alpha_1\\ \alpha_2\end{bmatrix} = 0 \tag{21}$$

where $-(N-1) \leq \alpha_1, \alpha_2 \leq (N-1)$ and $\alpha_1, \alpha_2 \neq 0$. For instance, if $N = 5$ and $\vec{S}_{1,2} = 6$ and $\vec{S}_{1,5} = 4$, then conditions $[\alpha_1 = 2]$ and $[\alpha_2 = -3]$ satisfy (21). (In one dimension, the vector spacings are positive or negative numbers.) Hence, there are data-input conflicts in input **X**. □

3.4 Constraints in Preloaded Data

If the velocity of a variable is zero, then the data corresponding to the variable have to be preloaded in the processors before computation begins. This problem involves designing a schedule that can overlap as much as possible the preloading of data with systolic computations without delaying these computations. A general approach is to decide when a particular stationary datum needs to be used in its first computation, and to develop a preloading schedule so that the bandwidth constraint of the processor array is satisfied and that the first computation can begin with the minimum delay. We like to point out a) that data do not have to be preloaded in any order that is governed by a dependence relation (as in systolic processing) as long as they do not conflict in using the inter-processor links, and the bandwidth of the input ports is not exceeded; b) that the optimal preloading schedule may depend on the velocities and data distributions of the moving data; and c) that preloading data may result in problem-size-dependent memory in each processor (a design alternative often disallowed in systolic arrays).

We discuss in Section 5 the effect of preloading data on computation/completion time for the transitive-closure problem.

4 Design Method

4.1 Formulation of the Search Problem

The design of a feasible processor array is equivalent to choosing an appropriate set of parameters in order to satisfy the constraints imposed by dependency and application requirements for a *specific* uniform recurrence equation and a *specific* problem size N. The search for the "best" design can be represented by the following optimization problem.

$$\textbf{Minimize}\quad b(N, t_1,\ldots,t_r,\vec{k}_1,\ldots,\vec{k}_r) \tag{22}$$

$$\textbf{Subject To:}\ \begin{cases}1 \leq t_i,\quad i = 1,\ldots,r,\\ 0 \leq \left|\vec{k}_i\right| \leq t_i,\quad i = 1,\ldots,r\\ \text{constraints defined in Theorems 1, 2, and 3}\\ \#\,PE \leq \#\,PE^{UB} \text{ and } T_c \leq T_c^{UB}\end{cases} \tag{23}$$

The objective function b defined in (22) is expressed in terms of attributes such as T_{comp}, computation time of the algorithm, T_{load}, load time for the initial inputs, T_{drain}, drain time for the final results, and $\#PE$, number of processing elements in the design. Note that the completion time of evaluating a recurrence is

$$T_c = T_{comp} + T_{load} + T_{drain} \tag{24}$$

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

The first two constraints in (23) follow directly from the definition of the parameters in GPM. Since the target array is systolic, displacement $|\vec{k}_i|$ should not exceed period t_i in order to prevent data broadcasting (velocities should not exceed one). In addition, the constraints $t_i \geq 1$, $i = 1, 2, \ldots, r$, mean that precedence constraints are satisfied.

The third constraint indicates that the recurrence is evaluated correctly by the processor array, satisfying dependency requirements (Theorems 1 and 2) and be free of data-link and computational conflicts (Theorem 3).

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

- Minimize T_c for a design with a maximum bound on $\#PE$, $\#PE^{UB}$;
- Minimize $\#PE$ for a design with a maximum bound on T_c, T_c^{UB}.

Both of these formulations represent trade-offs between T and $\#PE$.

The optimal design for the formulation given by (22) and (23) is found by a search algorithm. Since, in general, the objective function is nonlinear, involving functions such as ceiling, floor, and maximum/minimum of a set of terms, it is difficult to describe a comprehensive algorithm that covers all possible cases. In the rest of this section, we first describe a pruning strategy used in our search algorithm, followed by a discussion on 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 for special cases of optimizing T_c and $\#PE$.

4.2 Pruning Strategy

The search space defined by the constraints in (23) results in a worst-case complexity of

$$O\left(\sum_{i=1}^{g}\left(t_i^{\max}\right)^2\right) = O\left(\left(T_{comp}^{seq}\right)^{2g}\right), \tag{25}$$

where T_{comp}^{seq} is the time needed to process the recurrence sequentially, and t_i^{max} is the maximum value of period t_i such that the computation time $T_{comp} \leq T_{comp}^{seq}$. Equation (25) is true because we iterate in the worst case all combinations of t_i and $|\vec{k}_i| \leq t_i$, $i = 1, \ldots, r$. Note that this search space is polynomial in terms of the parameters in GPM and the size of the URE to be evaluated.

To reduce this search space, we need to develop effective pruning strategies so that suboptimal designs do not have to be evaluated. In this section, we present one such strategy that prunes suboptimal designs based on the incumbent design obtained so far in a search. Our pruning strategy takes the objective function b (assuming to be minimized) and decomposes it as follows:

$$b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) = f\left(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r, e(t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r)\right), \tag{26}$$

where N is not represented explicitly since it is a constant in the optimization. The decomposition is done in such a way that $e()$[2] is a monotonic function of its variables, which may be a subset of $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$. The intuition behind this decomposition is as follows.

If the objective function $b(t_1, \ldots, t_r, k_1, \ldots, k_r)$ is a monotonic function of its variables, then the optimal value of the parameters can be found by enumerating combinations of values of variables from their smallest permissible values (given by (23)) until a feasible design that satisfies Theorems 1, 2, and 3 is found. Since $b()$ is monotonic, the first feasible design obtained is also the optimal design.

The above idea of enumerating values of a monotonic function can be extended to the general case of nonmonotonic objective functions. This is done by first identifying $e()$, a monotonic component of the objective that can be enumerated efficiently. The search proceeds by enumerating designs so that values of $e()$ grow monotonically. (The combination of parameter values used in $e()$ are substituted into (23), and the constraint equations are solved to see if there exists a feasible design.) Whenever a feasible design is obtained, an upper bound on $e()$ is computed by setting variables in $b()$ that are not included in $e()$ to their extremum values. (This upper bound means that no optimal design will have an objective value whose monotonic component $e()$ is larger than the upper bound.) The search is then repeated, refining the upper bound each time a feasible design is found. It stops when the upper bound on $e()$ is smaller than or equal to $e()$ of the best feasible design.

For complex objective functions, rewriting the objective in terms of composite variables (expressed in terms of the primary variables $t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r$) can simplify the choice of the extremum values for variables other than those in $e()$. This is illustrated as follows:

Consider an objective expressed as a function of composite variables T_{comp}, T_{load}, T_{drain}, and $\#PE$ as follows.

$$B = b(T_{comp}, T_{load}, T_{drain}, \#PE). \tag{27}$$

It is easy to see that $T_{comp} = T_{comp}(t_1, \ldots, t_r)$ is monotonic with respect to the g periods $t_1, \ldots, t_r$. (An exact characterization is shown in Lemma 1 in Section 5.1 for the transitive-closure problem.) Hence, we choose T_{comp} as the monotonic component of objective function $b()$ and enumerate the periods $t_1, \ldots, t_r$ in an increasing order from their smallest permissible values (i.e., unity).

T_{comp}^{UB} can be refined if $b()$ is monotonically increasing

2. For notational ease, we denote functions without their arguments.

with T_{comp}, T_{load}, T_{drain} and $\#PE$. In this case, T^{UB}_{comp} can be obtained by setting $T_{load} = T_{drain} = 0$, $T_{comp} = T^{min}_{comp}$, and $\#PE = \#PE^{min}$ and solving

$$B^{inc} = b\left(T^{UB}_{comp}, T^{min}_{load}, T^{min}_{drain}, \#\,PE^{min}\right) \qquad (28)$$

$$= b\left(T^{UB}_{comp},\ 0,\ 0, \#\,PE^{min}\right) \qquad (29)$$

where B^{inc} is the objective value of the incumbent design obtained so far. Hence,

$$T^{UB}_{comp} = b^{-1}\left(B^{inc}, T^{min}_{load}, T^{min}_{drain}, \#\,PE^{min}\right) \qquad (30)$$

where $b^{-1}()$ is the inverse function of $b()$ that rearranges (29) to compute T^{UB}_{comp} in terms of known constants.

T^{UB}_{comp} can further be refined if $\#PE$ can be expressed as a function of $|\vec{k}_1|, \ldots, |\vec{k}_r|$. In this case, $\#PE$ is minimum when exactly one of the $|\vec{k}_i|$s is 1, and the rest of the $|\vec{k}_j|$, $j \neq i$, are 0. (An exact characterization is shown in Lemma 2 in Section 5.1 for the transitive-closure problem.)

For instance, if the objective function is

$$B = (T_{comp} + T_{load} + T_{drain})^2 \times \#PE. \qquad (31)$$

According to (29), we have

$$B^{inc} = (T_{comp} + 0 + 0)^2 \times \#\,PE$$

$$\Rightarrow T^{UB}_{comp} = \sqrt{B^{inc}/\#\,PE^{min}} \qquad (32)$$

Similarly, if the objective function to minimize completion time T_c,

$$B = T_c = T_{comp} + T_{load} + T_{drain}$$

$$\Rightarrow T^{UB}_{comp} = B^{inc} - (T^{min}_{load} + T^{min}_{drain}) - B^{inc} - (0+0) - B^{inc} = T^{inc}_c \qquad (33)$$

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

A special case of the optimization is to find a design with the minimum computation time T_{comp} (not including load and drain times). This was done in our earlier work [14], [16]. Here, $T^{UB}_{comp} = B^{inc} = T^{inc}_{comp}$, and the first feasible design is the optimal design that minimizes T_{comp}.

4.3 Search Procedure

In this section, we illustrate our search procedure by showing how we can minimize $b(\#PE, T_c) = b(T_{comp}, T_{load}, T_{drain}, \#PE)$ (27), 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, let these periods and displacements be t_i and $\vec{k}_i$, $1 \leq i \leq g$, respectively.
2) Initialize T^{UB}_{comp} to be T^{seq}_{comp}, 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 the 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 $(r - g)$ displacements using the conditions of Theorem 2.
6) Compute T^{cur}_{comp} using the periods and displacements found, where T^{cur}_{comp} is the computation time (without load and drain times) required for processing the recurrence. T^{cur}_{comp} is found by substituting the current values of t_i, $i = 1, \ldots, r$; an exact function is exemplified in Lemma 1 later. (Note that the design may not be feasible at this time.) If $T^{cur}_{compu} > T^{UB}_{comp}$, then exit with the incumbent design.
7) Solve for the spacing parameters from (13) defined in Theorem 1.
8) Check for data-input conflicts on the spacing parameters using Theorem 3; also, check whether the constraints on T_c and $\#PE$ are violated (Constraint 4 in (23)).
9) If the solution is not feasible, then increment one of the $|\vec{k}_i|$s and repeat Steps 5, 6, 7, and 8 until all $|\vec{k}_i|$ equal t_i, $i = 1, \ldots, r$. If all $|\vec{k}_i|$ equal t_i and no feasible design is found, then go to Step 10. If a feasible design is found, then go to Step 11.
10) Increment one of the periods such that T^{cur}_{compu} increases by the lowest possible value. Go to Step 4.
11) Compute B^{cur}, the objective value achieved by the current incumbent design found. If $B^{cur} < B^{inc}$, then set $B^{inc} = B^{cur}$, and compute T^{UB}_{compu} for the current design using (30). Increment one of the $|\vec{k}_i|$s so that T^{cur}_{compu} is increased by the lowest possible value. Go to Step 5.

For a design that minimizes $\#PE$, the search procedure described above needs to be changed. In this case, $e()$ should be defined as a function of $|\vec{k}_1|, \ldots, |\vec{k}_r|$, and the search should start iterating with the smallest combinations of $|\vec{k}_1|, \ldots, |\vec{k}_g|$. (See Lemma 2 in Section 5.1.)

5 Applications: Transitive Closure

Path-finding problems belong to an important class of optimization problems, which include computing transitive closures and shortest paths of a graph. 2D processor arrays for finding transitive closures have been presented before [17], [18]. In this section we synthesize a one-pass linear processor array for this problem using the Warshall-Floyd path-finding algorithm.

The transitive-closure problem is defined as follows.

Given an N-node directed graph with an $N \times N$ Boolean adjacency matrix $C[i, j]$, $1 \le i, j \le N$, the transitive closure $C^+[i, j] = 1$ if there exists a path from node i to node j, where $C[i, j] = 1$ if there is an edge from node i to node j or $i = j$, and $C[i, j] = 0$ otherwise. That is,

$$\begin{aligned}&\text{for } k, i, j = 1, N\\ &\mathbf{C}(i, j) = \mathbf{C}(i, j) + \mathbf{C}(i, k)\, \mathbf{C}(k, j) \qquad (34)\end{aligned}$$

The dependence structure of a general dynamic-programming formulation of the transitive-closure problem is irregular and difficult to map on a regularly connected planar processor array. To cope with this mapping problem, Kung et al. have converted the transitive-closure algorithm into an reindexed form and have mapped it to 2D spiral and orthogonal arrays [17]. Based on their algorithm, we obtain the following five dependence vectors after pipelining the variables.

$$\begin{aligned}
&\vec{d}_1 = (0,\, 0,\, 1)^t \text{ for } (k, i, j)^t \leftarrow (k, i, j-1)^t,\ 2 \le j \le N,\\
&\vec{d}_2 = (0,\, 1,\, 0)^t \text{ for } (k, i, j)^t \leftarrow (k, i-1, j)^t,\ 2 \le i \le N,\\
&\vec{d}_3 = (1,\, -1,\, -1)^t \text{ for } (k, i, k)^t \leftarrow (k-1, i+1, j+1)^t,\\
&\qquad 2 \le k \le N, 1 \le i, j \le N-1\\
&\vec{d}_4 = (1,\, -1,\, 0)^t \text{ for } (k, i, N)^t \leftarrow (k-1, i+1, N)^t,\\
&\qquad 2 \le k \le N,\ 1 \le i \le N-1,\\
&\vec{d}_5 = (1,\, 0,\, -1)^t \text{ for } (k, N, j)^t \leftarrow (k-1, N, j+1)^t,\\
&\qquad 2 \le k \le N,\ 1 \le j \le N-1 \qquad (35)
\end{aligned}$$

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

The key observation is as follows. Matrix **C** (whose transitive closure is to be found) is input along dependence vector $\vec{d}_3$. Inputs along other dependence vectors $\vec{d}_1, \vec{d}_2, \vec{d}_4, \vec{d}_5$ are nonexistent; i.e., they are never sent into the array from the external host. Hence, there are no data-input conflicts along these dependence directions. As a result, we need to consider data-input conflicts only along direction $\vec{d}_3$. Since dependencies $\vec{d}_3, \vec{d}_4$, and $\vec{d}_5$ never co-exist, 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 ($\vec{k}_1, \vec{k}_2, \vec{k}_3$), and two spacings ($\vec{S}_{3,1}, \vec{S}_{3,2}$). For a linear processor array, all the parameters are scalars. As derived in Example 4, the periods and velocities along directions $\vec{d}_4$ and $\vec{d}_5$ are given as $t_4 = t_1 + t_3$, $t_5 = t_2 + t_3$ (18), $\vec{k}_4 = \vec{k}_1 + \vec{k}_3$, and $\vec{k}_5 = \vec{k}_2 + \vec{k}_3$ (19), respectively. From Theorem 1 and (10), we get

$$\vec{S}_{3,1} = \frac{t_3\vec{k}_1 - t_1\vec{k}_3}{t_3},\quad \vec{S}_{3,2} = \frac{t_3\vec{k}_2 - t_2\vec{k}_3}{t_3} \qquad (36)$$

We illustrate in the rest of this section the following formulations of the optimization of linear processor arrays:

1) T_{comp}-optimal designs without bound on #*PE*;
2) T_c-optimal designs without bound on #*PE*;
3) #*PE*-optimal designs without bound on T_c or T_{comp}; and
4) optimal designs with specific bounds on T_{comp} or #*PE*; and
5) optimal designs with specific bounds on T_c or #*PE*.

5.1 Performance Attributes and Constraints

Before optimal designs can be found, we need to express performance attributes in the objective function in terms of the parameters in GPM. The attributes we are interested are T_{comp}, T_{load}, T_{drain}, #*PE*, and T_c, where $T_c = T_{load} + T_{comp} + T_{drain}$. In this section, we show three lemmas that express these performance attributes in terms of the parameters defined. We also show two constraints that refine the constraints defined in Theorem 3.

LEMMA 1. *The computation time T_{comp} without load and drain times for finding an $N \times N$ transitive closure is given by*

$$T_{comp} = (N-1)(2t_1 + 2t_2 + t_3) + 1 \qquad (37)$$

PROOF. The critical path in the execution is as follows:

$$\begin{aligned}&(1,1,1) \xrightarrow{(N-1)t_1} (1,1,N) \xrightarrow{(N-1)t_2} (1,N,N) \xrightarrow{(N-1)t_3}\\ &(N,1,1) \xrightarrow{(N-1)t_1} (N,1,N) \xrightarrow{(N-1)t_2} (N,N,N)\end{aligned}$$

Thus, T_{comp} is $(N-1)(2t_1 + 2t_2 + t_3) + 1$. □

LEMMA 2. *#PE, the number of processor for computing an $N \times N$ transitive closure on a linear processor array satisfying the dependencies in* (35), *is given by*

$$\#PE = (N-1)\left(\left|\vec{k}_1\right| + \left|\vec{k}_2\right| + \left|\vec{k}_1 + \vec{k}_2 + \vec{k}_3\right|\right) + 1 \qquad (38)$$

PROOF. See reference [13]. □

LEMMA 3. *Assuming that the input matrix is nonstationary, T_{load}, the load time, and T_{drain}, the drain time, for computing an $N \times N$ transitive closure on a linear processor array satisfying the dependencies defined in* (35) *are given by*

$$\begin{aligned}T_{load} &= T_{drain} =\\ &1 + (N-1)\left\lceil \frac{t_3\left\{\mathcal{G}(\vec{k}_1, \vec{k}_3) + \mathcal{G}(\vec{k}_2, \vec{k}_3) + \mathcal{G}[(\vec{k}_1 + \vec{k}_2 + \vec{k}_3), \vec{k}_3]\right\}}{\left|\vec{k}_3\right|} \right\rceil\\ &+ (N-1)\left[\mathcal{G}((\vec{S}_{3,1}), (-\vec{k}_3)) + \mathcal{G}((\vec{S}_{3,2}), (-\vec{k}_3))\right] \qquad (39)\end{aligned}$$

where

$$G(\vec{x}, \vec{y}) = \begin{cases}|\vec{x}| & \textit{if } \vec{x} \textit{ and } \vec{y} \textit{ are in opposite directions}\\ 0 & \textit{otherwise}\end{cases} \qquad (40)$$

PROOF. See reference [13]. □

Lemma 3 does not cover the case when the input matrix is stationary. As pointed out in Section 3.4, stationary inputs need to be preloaded in the processor array before computations begin. Since there is only one input matrix **C**, we assume that preloading takes a lower-bound time computed as the floor of the number of elements to be preloaded divided by the maximum number of input ports. A similar assumption is made when the final stationary results need to be drained. Even with this optimistic assumption,

TABLE 1
T_{COMP}-OPTIMAL LINEAR PROCESSOR ARRAYS
FOR FINDING TRANSITIVE CLOSURES OF $N \times N$ MATRICES

N	GPM: T_{comp}-Optimal Linear-Array Designs						
	Periods	Distances	Schedule	Allocation	Min T_{comp} Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	Π	**S**	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec.)
3	(1,1,2)	(0,−1,1)	(4,1,1)	(0,−1,0)	(5,13,5)	3	-
4	(1,1,3)	(0,−1,1)	(5,1,1)	(0,−1,0)	(10,22,10)	4	-
8	(1,1,5)	(0,−1,3)	(7,1,1)	(2,−1,0)	(15,64,15)	22	-
16	(1,2,5)	(0,−2,3)	(8,2,1)	(1,−2,0)	(61,166,61)	46	-
32	(1,3,6)	(0,−3,5)	(10,3,1)	(2,−3,0)	(125,435,125)	156	-
64	(1,5,7)	(0,−5,6)	(13,5,1)	(1,−5,0)	(379,1198,379)	379	-
100	(1,5,11)	(0,−5,9)	(17,5,1)	(4,−5,0)	(694,2278,694)	892	1
200	(1,8,13)	(1,−8,12)	(22,8,1)	(5,−8,1)	(1792,6170,1792)	2787	7
300	(1,9,18)	(0,−9,17)	(28,9,1)	(8,−9,0)	(2991,11363,2991)	5084	26

TABLE 2
T_C-OPTIMAL LINEAR PROCESSOR ARRAYS FOR FINDING TRANSITIVE CLOSURES OF $N \times N$ MATRICES

N	GPM: T_c-Optimal Linear-Array Designs						
	Periods	Distances	Schedule	Allocation	Min T_c Designs		SS10/30
	(t_1, t_2, t_3)	$(\vec{k}_1, \vec{k}_2, \vec{k}_3)$	Π	**S**	$(T_{load}, T_{comp}, T_{drain})$	#PE	(sec.)
3	(1,2,1)	(0,−1,1)	(4,2,1)	(0,−1,0)	(3,15,3)	3	-
4	(1,3,1)	(0,−1,1)	(5,3,1)	(0,−1,0)	(4,28,4)	4	-
8	(1,1,5)	(0,−1,3)	(7,1,1)	(2,−1,0)	(15,64,15)	22	-
16	(1,2,6)	(0,−1,5)	(9,2,1)	(4,−1,0)	(31,181,31)	76	-
32	(1,2,9)	(0,−2,7)	(12,2,1)	(5,−2,0)	(94,466,94)	218	1
64	(1,5,10)	(0,−2,9)	(16,5,1)	(7,−2,0)	(190,1387,190)	568	5
100	(1,4,15)	(0,−3,14)	(20,4,1)	(11,−3,0)	(397,2476,397)	1387	14
200	(6,1,19)	(−5,0,18)	(26,1,6)	(13,0,−5)	(1195,6568,1195)	3583	91
300	(1,7,24)	(0,−6,23)	(32,7,1)	(17,−6,0)	(2094,11961,2094)	6878	265

we did not find any design with stationary inputs/outputs that out-perform designs with moving inputs. Although this observation is not true in general, we like to point out that a schedule to preload data in the processor array needs not be governed by the data dependence relations, and that a general preloading schedule may depend on specific design parameters (such as values of the GPM parameters) and architectural constraints (such as bandwidth and memory).

For linear-array synthesis, since the spacings are scalars, let $s_{3,1}$ be $|\vec{S}_{3,1}|$ and $s_{3,2}$ be $|\vec{S}_{3,2}|$. In addition, the condition for data-input conflict (Theorem 3) can be refined as follows.

THEOREM 4. *Data-input conflicts occur in the N-by-N input matrix* **C** *if and only if*

$$\frac{s_{3,1}}{\xi} < N, \text{ and } \frac{s_{3,2}}{\xi} < N \tag{41}$$

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

PROOF. See reference [13]. □

COROLLARY 2. *For any feasible design,* $s_{3,1} + s_{3,2} \geq N + 1$.

PROOF. Assume for contradiction that $s_{3,1} + s_{3,2} = x$, $x < N + 1$. Then $1 \leq s_{3,1}, s_{3,2} \leq (x - 1)$. If ξ is $GCD(s_{3,1}, s_{3,2})$, then

$$1 \leq \frac{s_{3,1}}{\xi}, \frac{s_{3,2}}{\xi} \leq (x - 1) \leq N$$

According to Theorem 4, data-input conflicts are present, and the solution is not feasible. □

5.2 Time-Optimal and Processor-Optimal Linear-Array Designs

Table 1 shows the optimal linear-array designs found by the search procedure of GPM (see Section 4.3) in which the objective is to minimize T_{comp} (computation time, not including load and drain times) without bounds on *#PE*. In finding these designs, t_3 is incremented before t_1 or t_2 in Step 10 of the search procedure. This is done as it increases T_{comp} by the smallest amount. Among all the designs that have the minimum T_{comp}, we found designs that require the minimum *#PE*, followed by finding designs that require the minimum T_{load} and T_{drain}. We list T_{load}, T_{comp}, T_{drain}, *#PE*s needed, and the CPU time used by the search procedure running on a Sun Sparcstation 10/30. We also list the equivalent values of schedule vector Π and allocation matrix **S** of DM by solving (9) and (11).

In a similar way, we find designs that optimize T_c (completion time, including load and drain times) without bounds on *#PE*. (See Table 2.) Note that these designs have less total completion time and more *#PE*s than the corresponding designs in Table 1. 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 one optimizing T_{comp}. We also list the equivalent Π and **S** in DM for minimizing T_c.

Our results in Tables 1 and 2 demonstrate that GPM, based on the equivalence between GPM and DM as shown

TABLE 3
PARAMETERS USED IN LK, SF, AND GPM ON #PE-OPTIMAL LINEAR PROCESSOR ARRAYS FOR FINDING TRANSITIVE CLOSURES OF $N \times N$ MATRICES

Method	$\vec{\Pi}$	$\mathbf{S}$	(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	$[N+1, 1, 1]^t$	$[0, 0, -1]^t$	$(1, 1, N-1)$	$(-1, 0, 1)$

TABLE 4
#PE-OPTIMAL LINEAR PROCESSOR ARRAYS FOR FINDING TRANSITIVE CLOSURES OF $N \times N$ MATRICES (PARAMETERS FOR GPM ARE SHOWN IN THEOREM 5 AND IN TABLE 3)

	Designs by LK [7]		Designs by SF [6]		Designs by GPM	
N	$(T_{load}, T_{comp}, T_{drain})$	#PE	$(T_{load}, T_{comp}, T_{drain})$	#PE	$(T_{load}, T_{comp}, T_{drain})$	#PE
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

in (9) and (11), can serve as a powerful tool to find optimal designs in DM.

It is important to point out that the objective used (whether to minimize T_{comp} or 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.

If the objective is to minimize #PE in the linear processor array, then Theorem 5 characterizes the #PE-optimal design.

THEOREM 5. *The combinations of parameters* $(t_1, t_2, t_3) = (1, 1, N-1)$ *and* $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, \pm 1, \mp 1)$ *or* $(\pm 1, 0, \mp 1)$ *result in linear processor arrays with a primary objective of minimizing the number of PEs, and a secondary objective of minimizing the computation time.*

Proof. See reference [13]. □

Table 4 shows the #PE-optimal designs obtained by GPM as well as those obtained by Lee and Kedem (LK) [7] and Shang and Fortes (SF) [6]. In this table, we show the load and drain times, computation times, and #PEs for designs derived by these three methods. $\vec{\Pi}$, $\mathbf{S}$, and the corresponding parameters in GPM are summarized in Table 3.

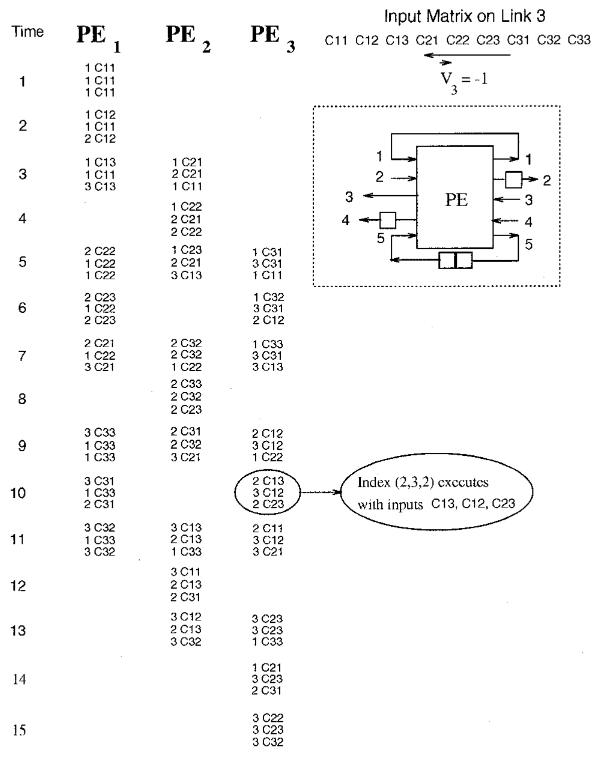
Table 4 shows that both the SF and GPM designs require the minimum number of PEs. The SF designs, however, were developed based on different assumptions. According to Lemma 1 and Table 3, the SF designs have a computation time $T_{comp} = (N-1)(N+2)+1$. This computation time is lower than that of the GPM designs characterized by Theorem 5. This difference is attributed to the fact that Shang and Fortes assumed that contention must only be avoided after the first use of a variable 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 post-drained. In GPM, we consider both contentions in computations as well as in data links. Excluding designs that have these contentions results in designs that require slightly longer load, drain, and computation times.

To illustrate the point above, we compute using (36) the spacings used in the SF design [6]: $s_{3,1} = -(N-1)/(N-2)$ and $s_{3,2} = -1/(N-2)$). These values of spacings result in data-input conflicts between tokens ($\mathbf{C}_{1,j}$ and $\mathbf{C}_{N,j-1}$), $j = 2, 3, \ldots, N$, of input matrix $\mathbf{C}$ (Theorem 4).

The space-time diagrams of a linear processor array optimizing T_c for $N = 3$ is shown in Fig. 2. This design has parameters $(t_1, t_2, t_3) = (1, 2, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. It minimizes both T_c and #PE, and, therefore, minimizes any objective of the form $\# PE^x \cdot T_c^y$ for $x, y \geq 1$. Note that the load and drain times are not shown in these disgrams. Further, 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 assignments are given in Tables I and II in reference [8].

5.3 Processor-Time Trade-Offs

Comparing the results in Tables 2 and 4, we found, for instance, that for a problem of size of 200, the T_c-optimal design is 13.35 times faster than the #PE-optimal design in terms of completion time, and uses 17.9 times more PEs than the #PE-optimal design. (The T_c-optimal design for $N = 200$ requires 8,958 time units and 3,583 PEs, whereas the #PE-optimal design requires 119,602 time units and 200 PEs.) 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, a possible objective could be to have as few proces-



Time	PE_1	PE_2	PE_3
1	1 C11 1 C11 1 C11		
2	1 C12 1 C11 2 C12		
3	1 C13 1 C11 3 C13	1 C21 2 C21 1 C11	
4		1 C22 2 C21 2 C22	
5	2 C22 1 C22 1 C22	1 C23 2 C21 3 C13	1 C31 3 C31 1 C11
6	2 C23 1 C22 2 C23		1 C32 3 C31 2 C12
7	2 C21 1 C22 3 C21	2 C32 2 C32 1 C22	1 C33 3 C31 3 C13
8		2 C33 2 C32 2 C23	
9	3 C33 1 C33 1 C33	2 C31 2 C32 3 C21	2 C12 3 C12 1 C22
10	3 C31 1 C33 2 C31		2 C13 3 C12 2 C23
11	3 C32 1 C33 3 C32	3 C13 2 C13 1 C33	2 C11 3 C12 3 C21
12		3 C11 2 C13 2 C31	
13		3 C12 2 C13 3 C32	3 C23 3 C23 1 C33
14			1 C21 3 C23 2 C31
15			3 C22 3 C23 3 C32

Input Matrix on Link 3
C11 C12 C13 C21 C22 C23 C31 C32 C33
$V_3 = -1$
PE
1
2
3
4
5
1
2
3
4
5
Index (2,3,2) executes with inputs C13, C12, C23

Fig. 2. Linear processor array for transitive closure of a 3×3 matrix with $(t_1, t_2, t_3) = (1, 2, 1)$ and $(\vec{k}_1, \vec{k}_2, \vec{k}_3) = (0, -1, 1)$. The array is optimal for minimum T_c, minimum $\#PE$, and minimum $\# PE^x \times T_c^y$, x, $y \geq 1$. The PE used is the same as in Lee and Kedem's design.

sors as possible, so long as the time is within a preset upper limit, T_c^{ub} (or T_{comp}^{ub}), or to minimize T_c (or T_{comp}) with $\#PE$ less than a given upper bound $\#PE^{ub}$.

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 $\#PE \geq \#PE^{max}$ would not be useful as their computation times have to be at least T_{comp}^{min}. On the other hand, let T_{comp}^{max} and $\#PE^{min}$ be, respectively, the computation time and #PE of the minimum-processor design (from Theorem 5 and Lemma 2, $\#PE^{min} = N$). Again, there is no benefit in obtaining designs with $T_{comp} \geq T_{comp}^{max}$, as the number of PEs cannot be reduced below $\#PE^{min}$. In this case, we are interested to find designs with computation time greater than T_{comp}^{min} and $\#PE$ less than $\#PE^{max}$.

Fig. 3 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 results of 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. These curves are stepped

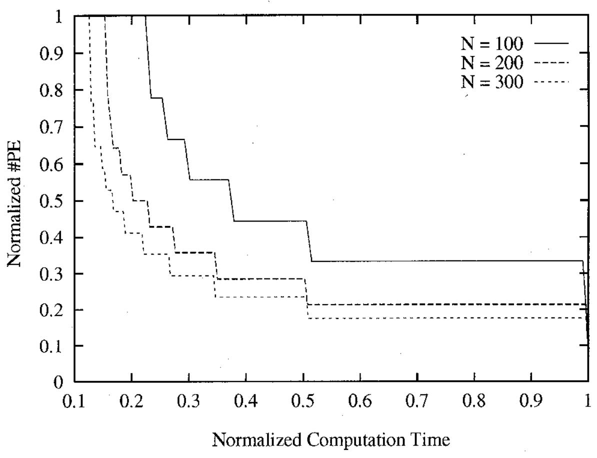


Fig. 3. Performance trade-offs: Variation in (a) minimum $\#PE$ with time bound T_{comp}^{ub} and (b) minimum T_{comp} with processor bound $\#PE^{ub}$. The plots are shown for N = 100, 200, and 300.

because there are only a small and finite number of processor-array configurations that can satisfy the given time constraints. If the goal is to find $\#PE$-optimal designs, then we will have a small number of array configurations; for each configuration, we will select the one with the minimum computation time.

Given the bound T_{comp}^{ub} (*resp.*, $\#PE^{ub}$) the designer can use Fig. 3 to find the minimum $\#PE$ (*resp.*, T_{comp}) required, and decide (possibly from a cost perspective) if it is acceptable. Again, the designer can 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, then the design with the minimum $\#PE$ for a given time bound is determined using Fig. 3. First, a horizontal line is drawn across the graph for the desired bound on $\#PE$. 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}, then a feasible design can be obtained by the procedure discussed in Section 4.3. This now represents the best design under both time and processor constraints.

Another observation from Fig. 3 is that the plots for larger N decrease more rapidly than those for smaller N. Hence, for larger N, there is a substantial reduction in $\#PE$ (*resp.*, T_{comp}) for a relatively small increase of the computation time (*resp.*, $\#PE$) from the optimum. Hence, for large N, there are more attractive alternatives than the time- or $\#PE$-optimal designs.

Fig. 4 shows a similar plot as in Fig. 3 except that we depict the difference between trade-offs obtained on T_c and $\#PE$ versus trade-offs obtained on T_{comp} and $\#PE$. Two sets of curves are shown, one for designs that minimize T_{comp}, and the other for designs that minimize T_c, for N equal to 100 and 200, respectively. The y-axis of these curves is normalized with respect to $\#PE$ when T_c is minimum (since these de-

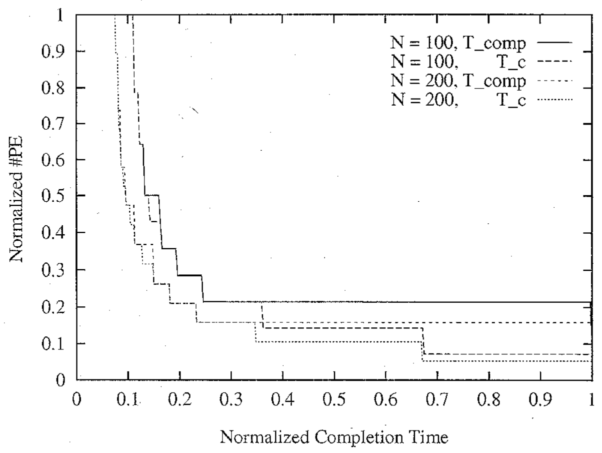


Fig. 4. Performance trade-offs: Variations in *#PE* with T_c and *#PE* with T_{comp}. The plots are given for two problem sizes, $N = 100$ and 200.

signs require more PEs and less T_c), and the x-axis is normalized with respect to T_c when $T_{comp} = T_{comp}^{max}$. These graphs show the difference between designs obtained by different objectives. Given a bound T_c^{ub}, 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}.

6 Final Remarks

Algorithm-specific processor arrays can be systematically designed with the help of the general parameter-based approach (GPM) discussed in this paper. This work generalizes the initial work on a parameter based approach [10] by making three important and nontrivial extensions.

1) We consider the recurrence model as a general n-D recurrence with arbitrary constant dependence vectors instead of a specific 3D one. The target processor arrays are also allowed to be of any lower dimension m, where $1 \leq m \leq n$. We provide new necessary conditions to guarantee the correctness of systolic processing in mapping high-dimensional recurrences to lower-dimensional processor arrays. These conditions define a search space polynomial in complexity with respect to the size of the recurrence to be mapped. In contrast, previous methods for finding optimal designs are based on integer linear programming with a search space of exponential complexity.

TABLE 5
Comparison Between Dependency-Based and Parameter-Based Methods [12]

Feature	Dependency-Based Method	Generalized Parameter Method
Applicable Recurrences	General and applicable to uniform as well as non-uniform recurrences.	Homogeneous uniform recurrences or uniformized affine recurrences.
Representation	Schedule Vector and Allocation Matrix: they are represented in the Cartesian coordinate system with unit vectors as basis vectors; for the dimension-reduction technique [6], the mappings are rank-deficient; (*i.e.*, $\vec{\Pi}$ and $\mathbf{S}$ yield $\mathbf{T}$ where $rank(\mathbf{T}) \leq n$).	Periods and Displacements: they are represented in a possibly non-orthogonal coordinate system with dependence vectors as basis vectors; hence, for uniform recurrences, the representations in DM and GPM are equivalent and are derivable from each other by a coordinate (linear) transformation.
Characteristics of controls in processor array	Non-uniform in the general case by specifying a general processor allocation matrix; processor arrays derived may have in the general case 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	Computation-time optimal designs or processor-optimal designs with linear objective function and linear constraints.	General non-linear objective function and constraints with certain monotonicity properties on the objective function; new constraints have been developed that avoid data-link conflicts.
Search methods for finding processor array designs	Choose heuristically a processor allocation matrix, and find schedule vectors satisfying processor-allocation constraints; methods for 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 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.	Trade-offs between number of processors and computation time, or between number of processors and completion time (including load and drain times) for a specific problem instance can be obtained.
Summary	The two methods are equivalent representations for synthesizing uniform recurrences. The formulation of the design optimization problem and the search techniques developed are equally applicable in both representations.	

2) We extend our search method to handle general non-linear objectives that may vary non-monotonically with the parameters, and introduce new pruning strategies to prune suboptimal designs in the search space so that optimal designs can be found efficiently. We show a) optimal designs that include load and drain times in the objective (which introduce non-linearity in the objective function and constraints), and b) optimal designs with constraints on number of allowable processing elements and/or completion time. Such designs cannot be found by previous methods.
3) We show the equivalence between DM and GPM by providing necessary equations to transform parameters used in DM to those used in GPM, and vice versa. The equivalence allows the designers familiar with DM to utilize the efficiency of GPM to find optimal designs.

Table 5 summarizes the unique features of GPM and DM.

Acknowledgments and References to Earlier Papers

The authors are grateful to Prof. Jose Fortes, Wei-jia Shang, and Michael Loui for commenting on earlier versions of this work.

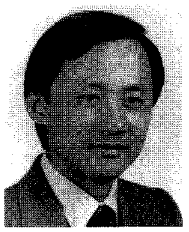
Research Supported by National Science Foundation grants MIP 88-10584 and MIP 92-18715, Joint Services Electronics Program contract N00014-90-J-1270, and an IBM graduate fellowship grant.

References [14], [16] describe preliminary versions of of Theorems 1, 2, and 3. They show optimization of linear arrays for computing transitive closures [14] and matrix multiplication [16] but did not show trade-off analysis and the relationship of GPM to DM. More comprehensive results comparing DM and GPM were shown in reference [12]. Applications of the method presented in this paper on the matrix-multiplication problem were shown in reference [19]. Complete proofs of all the theorems can be found in Dr. Ganapathy's PhD thesis [13]. Finally, electronic copies of papers written by the authors can be obtained through the World-Wide Web (http://manip.crhc.uiuc.edu).

References

[1] R.H. Kuhn, "Optimization and Interconnections Complexity for Parallel Processors, Single Stage Networks, and Decision Trees," PhD dissertation, Dept. of Computer Science, Univ. of Illinois, Urbana, Oct. 1980.
[2] D.I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. Computers*, vol. 31, no. 11, pp. 1,121-1,126, Nov. 1982.
[3] H.T. Kung, "Why Systolic Architectures?" *Computer*, vol. 15, no. 1, pp. 37-46, Jan. 1982
[4] J.A.B. Fortes, K.-S. Fu, and B.W. Wah, "Systematic Design Approached for Algorithmically Specified Systolic Arrays," *Computer Architecture: Concepts and Systems*, V.M. Milutinovic, ed., pp. 454-494. North Holland, 1988.
[5] Z. Chen and W. Shang, "On Uniformization of Affine Dependence Algorithms," *Proc. Fourth Symp. Parallel and Distributed Systems*, vol. 3, pp. 128-137, Dec. 1992.
[6] W. Shand and J.A.B. Fortes, "On Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 5, pp. 350-363, May 1992.
[7] P.-Z. Lee and Z.M. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 1, pp. 64-76, Jan. 1990.
[8] P.-Z. Lee and Z.M. Kedem, "Synthesizing Linear Array Algorithms from Nested for Loop Algorithms," *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1,578-1,597, Dec. 1988.
[9] V.P. Roychowdhury and T. Kailath, "Subspace Scheduling and Parallel Implementation of Non-Systolic Regular Iterative Algorithms," *J.VLSI Signal Processing*, vol. 1. Kluwer Academic, 1989.
[10] G.-J. Li and B.W. Wah, "The Design of Optimal Systolic Arrays," *IEEE Trans. Computers*, vol. 34, no. 1, pp. 66-77, Jan. 1985.
[11] M.T. O'Keefe, J.A.B. Fortes, and B.W. Wah, "On the Relationship Between Systolic Array Design Methodologies," *IEEE Trans. Computers*, vol. 41, no. 12, pp. 1,589-1,593, Dec. 1991.
[12] J.A.B. Fortes, B.W. Wah, W. Shang, and K.N. Ganapathy, "Algorithm-Specific Parallel Processing with Linear Processor Arrays," *Advances in Computers*, M. Yovits, ed. Academic Press, 1994.
[13] K. Ganapathy, "Mapping Regular Recursive Algorithms to Fine-Grained Processor Arrays," PhD dissertation, Univ. of Illinois, Urbana-Champaign, Apr. 1994.
[14] K.N. Ganapathy and B.W. Wah, "Synthesizing Optimal Lower Dimensional Processor Arrays," *Proc. Int'l Conf. Parallel Processing*, pp. 96-103. Pennsylvania State Univ. Press, Aug. 1992.
[15] J. Zue, "A New Formulation of the Mapping Conditions for the Synthesis of Linear Systolic Arrays," *Proc. Application Specific Array Processors*, pp. 297-308. IEEE CS Press, 1993.
[16] K.N. Ganapathy and B.W. Wah, "Optimal Design of Lower Dimensional Processor Arrays for Uniform Recurrences," *Proc. Application Specific Array Processors*, pp. 636-648. IEEE CS Press, Aug. 1992.
[17] S.Y. Kung, S.C. Lo, and P.S. Lewis, "Optimal Systolic Design for Transitive Closure and Shortest Path Problems," *IEEE Trans. Computers*, vol. 36, no. 5, pp. 603-614, May 1987.
[18] G. Rote, "A Systolic Array for Algebraic Path Problem," *Computing*, vol. 34, pp. 192-219. Springer-Verlag, 1985.
[19] K.N. Ganapathy and B.W. Wah, "Designing a Coprocessor for Regular Recurrent Computations," *Proc. Fifth IEEE Symp. Parallel and Distributed Systems*, pp. 806-813, Dec. 1993.

Kumar Ganapathy received the BTech degree in electrical engineering from the Indian Institute of Technology, Madras, in 1987, the MS degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1990, and the PhD degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1994. After graduation, he joined the Digital Communication Division at Rockwell International, Newport Beach, California, where he works on high-speed DSP archi-tectures, low-power designs, and other aspects of VLSI signal processing.



Benjamin W. Wah received his PhD degree in computer science from the University of California, Berkeley, in 1979. He is currently a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. He previously served on the faculty of Purdue University (1979-1985), as a program director at the National Science Foundation (1988-198), as Fujitsu Visiting Chair Professor of Intelligence Engineering, University of Tokyo (1992), and McKay Visiting Professor of Electrical Engineering and Computer Science, University of California, Berkeley (1994). In 1989, he was awarded a University Scholar of the University of Illinois. His current research interests are in the areas of parallel and distributed processing, artificial intelligence, and nonlinear optimization.

Dr. Wah is editor-in-chief of *IEEE Transactions on Knowledge and Data Engineering*, and serves on the editorial boards of the *Journal of Parallel and Distributed Computing*, *Information Sciences*, *International Journal on Artificial Intelligence Tools*, and *Journal of VLSI Signal Processing*. He has chaired a number of conferences, including the 1995 Computer Software and Applications Conference and the 1996 International Conference on Neural Networks. He has served in the IEEE Computer Society as a member of its Board of Governors and is currently serving on the IEEE CS Publications Board, Press Activities Board, and Fellows Committee.