Parallel Processing Letters Vol. 4 No. 3 (1994) 339–350
© World Scientific Publishing Company

OPTIMAL SYNTHESIS OF PROCESSOR ARRAYS WITH PIPELINED ARITHMETIC UNITS*

KUMAR GANAPATHY and BENJAMIN W. WAH
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
1308 West Main Street
Urbana, IL 61801, U.S.A.
E-mail: {*kumar,wah*}*@manip.crhc.uiuc.edu*

Received 17 July 1993
Revised 25 November 1993
Accepted by Y. Robert

ABSTRACT

Two-level pipelining in processor arrays (PAs) involves pipelining of operations across processing elements (PEs) and pipelining of operations in functional units in each PE. Although it is an attractive method for improving the throughput of PAs, existing methods for generating PAs with two-level pipelining are restricted and cannot systematically explore the entire space of feasible designs. In this paper, we extend a systematic design method, called General Parameter Method (GPM), we have developed earlier to find optimal designs of PAs with two-level pipelines. The basic idea is to add new constraints on periods of data flows to include the effect of internal functional pipelines in the PEs. As an illustration, we present pipelined PA designs for computing matrix products. For n-dimensional meshes and other symmetric problems, we provide an efficient scheme to obtain a pipelined PA from a non-pipelined PA using a reindexing transformation. This scheme is used in GPM as a pruning condition to arrive at optimal pipelined PAs efficiently. For pipelines with minimum initiation interval (MII) greater than unity, we show additional constraints that ensure correctness of the synthesized PAs.

1. Introduction

In this paper, we present a method for synthesizing *processor arrays* (PAs) with *pipelined functional units* (PFU). A VLSI PA has simple *processing elements* (PE) that are regularly connected and globally synchronized, and have implicit control within each. Its advantages are cost-effective VLSI implementations and alleviation of the *von-Neumann* memory bottleneck. PAs have been applied in numerous applications in real-time signal processing and as hardware libraries for general-purpose computers.

Given the design of a PA, its performance can be improved by increasing the rate at which the PA is clocked. However, it has become increasingly difficult to reduce the cycle time of non-pipelined arithmetic-logic units (ALUs) in the PEs beyond

*Research Supported by Joint Services Electronics Program contract N00014-90-J-1270, National Science Foundation grant MIP 92-18715, and an IBM Graduate Fellowship Grant.

a certain limit. A possible solution is to design the ALUs as PFUs, and subdivide computations performed by each ALU into multiple stages. This technique of pipelining operations across PEs, and pipelining operations in the functional units of the PEs, is known as *two-level pipelining.*

Using two-level pipelining, it is much easier to design a PA with the same clock rate, using PFUs that perform fewer operations per cycle than a PA with non-pipelined functional units. This leads to designs that require less power and are easier to manufacture. On the other hand, the pipelined design can be clocked at a much higher rate using superpipelining (upto S times faster for an S-stage pipeline than a non-pipelined design since there is less circuitry in each pipeline stage). This leads to a higher throughput than a non-pipelined design.

A major problem in designing two-level pipelines is the lack of a systematic method that evaluates alternative designs and finds optimal designs with respect to user-specified objective function and design constraints. The search for the optimal design is different than that of PAs without PFUs because internal pipelining introduces a new dimension in the design problem (effectively constraining the search space). Further, internal pipelines in a PE can have a large number of stages (say over 20 stages) when several functional units in a PE are chained together to perform the operations in the body of a complex inner loop.

This paper is motivated by the problems described above and presents a systematic method for designing two-level (deeply) pipelined PAs that optimize user-specified objective and design constraints for a given application and problem size. We present the enhancement of a systematic design method, called General Parameter method (GPM), we have developed earlier [1,2] to map nested-loop algorithms into PAs with PFUs. Our enhanced GPM can find optimal PA designs efficiently using a polynomial-time search algorithm.

This paper is organized as follows. In Section 2, we summarize the features of GPM and its search method [1,2]. We then present new constraints in Section 2.4 that can be incorporated into GPM to automatically synthesize PAs with PFUs from the algorithm description. Such a systematic method to generate two-level pipelined designs has not been proposed before. In Section 3, we show a closed-form condition for deriving pipelined designs from non-pipelined ones for n-dimensional (written as n-D) mesh algorithms. This condition can also be used in GPM for pruning suboptimal designs. In Section 4, we present the results of applying the enhanced GPM to the matrix-product application. Comparison to previous work are given in Section 5, and conclusions are drawn in Section 6.

2. General Parameter Method

2.1. Recurrences

The algorithm description used in GPM is in the form of recurrence equations. An n-D recurrence is equivalent to a collection of n-nested loops represented as a dependence graph (DG), with arcs denoting the loop-carried dependencies and nodes (also referred to as index points) denoting the body of the innermost loop. In this paper, we focus on algorithms with constant dependencies equivalent to a set of *uniform recurrence equations.* These include algorithms that can be modeled directly as uniform recurrences as well as *affine recurrences* that can be "uniformized" to yield equivalent sets of *uniform recurrences*. The loop bounds are affine functions of the loop variables and define a convex polyhedral domain. Branch statements, if any, must involve only loop variables and do not go outside the loop containing the branch statement *i.e.*, the unit for scheduling is the body of the loops enclosing all the branch statements.

Example 1. The following collection of equations are the uniform recurrences for the product of two $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ to yield a resulting matrix $\mathbf{C}$.

$$\begin{aligned} C(i,j,k) &= C(i,j,k-1) + A(i,j,k) \times B(i,j,k) \\ A(i,j,k) &= A(i,j-1,k) \\ B(i,j,k) &= B(i-1,j,k) \end{aligned}$$

The boundary conditions are $A(i,0,k) = \mathbf{A}[i,k], B(0,j,k) = \mathbf{B}[j,k], C(i,j,0) = \mathbf{C}[i,j]$. There are three dependencies, $(0,0,1)^t$, $(0,1,0)^t$, and $(1,0,0)^t$, in the algorithm.

2.2. Parameters

GPM synthesizes a set of n-D uniform recurrence equations into a set of vector and scalar parameters that characterize the behavior, correctness, and performance of an m-D PA, where $m < n$. When a uniform recurrence is executed in a PA, the computations are periodic and equally-spaced in the PA. GPM captures this periodicity by a minimal set of parameters defined as follows [1,2].

Parameter 1: Period. A period captures the time between execution of the source and sink index points of a dependence vector. Suppose $\tau_c(\vec{I})$ is the time at which index point $\vec{I}$ (a n-D vector denoting a node in the DG) is executed. We define the period of computation t_j along dependence direction $\vec{d_j}$ as

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

The number of periods defined is equal to r, the number of dependencies in the algorithm.

Parameter 2: Velocity. Velocity $\vec{V_j}$ of data flowing along dependence vector $\vec{d_j}$ is defined as the directional distance traversed in one clock cycle. All data tokens of a variable have the same velocity (both speed and direction) which is constant during the execution in the PA. The total number of velocity parameters is r (one for each variable), with each velocity being an m-element vector, where m is the dimension of the PA. Hence, velocity, $\vec{V_j}$ is given by

$$\vec{V_j} = \frac{\vec{k_j}}{t_j}, \quad j = 1, 2, \cdots, r, \tag{2}$$

where $\vec{k_j}$ is the (vector) distance between the execution locations of the source and sink index points of dependence vector $\vec{d_j}$.

Parameter 3: Spacing or Data Distribution. Consider variable Ω_i pipelined along vector $\vec{d_i}$. 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 a vector $\vec{d_i}$, $1 \le i \le r$, there are $r-1$ non-trivial spacing parameters for each variable and one trivial spacing parameter, $\vec{S}_{i,i} = \vec{0}$. Each spacing parameter $\vec{S}_{i,j}$ is an m-D vector, where m is the dimension of the PA.

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 displacements (or velocities), and r^2 are spacings (r of these spacings are trivially zero).

2.3. Design Method

The search for the "best" design can be represented by the following optimization problem.

$$\textbf{Minimize} \quad b(N, t_1, \ldots, t_r, \vec{k}_1, \ldots, \vec{k}_r) \tag{3}$$

$$\textbf{Subject To:} \quad \begin{cases} 1 \le t_i, \quad i = 1, \ldots, r, \\ 0 \le \left|\vec{k}_i\right| \le t_i, \quad i = 1, \ldots, r \\ \text{constraints defined in Eq's 5, 6 and 7} \\ \#PE \le \#PE^{UB} \text{ and } T_c \le T_c^{UB}. \end{cases} \tag{4}$$

The constraints on the parameters for any feasible non-pipelined lower-dimensional PA are listed below.

1. All data tokens participating in computing the result at an index point must arrive at a PE simultaneously.

$$\vec{V}_i\, t_i = \vec{V}_j\, t_i + \vec{S}_{j,i}, \quad i,\, j = 1, 2, \cdots, r. \tag{5}$$

2. Linearly dependent dependence vectors must satisfy the following constraint:

$$\vec{T} \cdot \mathbf{N} = 0, \text{ and } \mathbf{K}\,\mathbf{N} = 0\,, \tag{6}$$

where g is the rank of dependency matrix $\mathbf{D}$, and $\mathbf{N}$ is the Null Space of $\mathbf{D}$.

3. Data-input conflicts (two tokens sent into a boundary PE simultaneously) must be avoided. Let $\mathbf{S}_i' = \left[\vec{S}_{i,1}, \vec{S}_{i,2}, \ldots, \vec{S}_{i,g-1}\right]$ be a matrix of $g-1$ independent spacings. Further, let $L_k, U_k, k = 1, 2, \cdots, g-1$, be integers such that the positions of all tokens of variable Ω_i, relative to a common origin, are defined as $\sum_{k=1}^{g-1} \vec{S}_{i,k}\beta_k$, $L_k \le \beta_k \le U_k$. To avoid data-input conflicts, we need the following constraint.

$$\mathbf{S}_i'\,\vec{\alpha} \ne \vec{0}, \text{ where } \vec{\alpha} \ne \vec{0}, \text{ and } \alpha_i \in [(L_i - U_i), \cdots, (L_i + U_i)], 1 \le i \le g-1. \tag{7}$$

The target PA can be obtained by first specifying the desired objective function $b()$, and by searching the solution space for the appropriate design that optimizes the given objective while satisfying the above constraints. The search for the feasible (or optimal) design is based on the following observations.

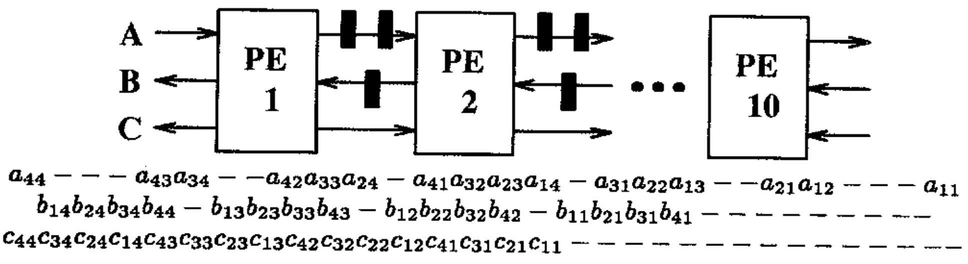
A
B
C
PE 1
PE 2
PE 10
$a_{44} - - - a_{43}a_{34} - -a_{42}a_{33}a_{24} - a_{41}a_{32}a_{23}a_{14} - a_{31}a_{22}a_{13} - -a_{21}a_{12} - - - a_{11}$
$b_{14}b_{24}b_{34}b_{44} - b_{13}b_{23}b_{33}b_{43} - b_{12}b_{22}b_{32}b_{42} - b_{11}b_{21}b_{31}b_{41} - - - - - - - - - -$
$c_{44}c_{34}c_{24}c_{14}c_{43}c_{33}c_{23}c_{13}c_{42}c_{32}c_{22}c_{12}c_{41}c_{31}c_{21}c_{11} - - - - - - - - - - - - - - - -$

Fig. 1. Linear PA for 4 x 4 matrix product with non-pipelined functional units.

a) The objective of a design at the logical level is a function of the number of PEs and the completion time of processing an application problem. The "optimal" design is *problem-size dependent.*

b) It is possible to express the number of PEs used and the completion time as functions of the parameters defined in GPM. Examples of such functions for computing matrix products [2] and transitive closures [1] can be found in the references.

c) If the objective is a monotonic function of the parameters defined in GPM, then the search space can be ordered in such a way that the first feasible design found is the optimal design. Such is the case when minimizing the completion time or the number of PEs in the matrix-multiplication [2] and transitive-closure [1] problems. The search procedure is very efficient because the search space is a polynomial function of the problem size and can be ordered in its traversal.

d) If the objective is a non-monotonic function of the parameters defined in GPM, then it is difficult to order the search space as is in the case of monotonic objective functions. In this case, a monotonic component of the objective is identified and enumerated efficiently. The difference, however, is that a feasible solution defines an upper bound on the value of the monotonic component being enumerated. The search is then continued to find better designs until the upper bound is reached. A special case happens when the objective function is monotonic. In this case, the upper bound defined by the feasible solution is equal to the feasible objective value itself, making the feasible solution optimal.

e) Additional design constraints can be incorporated easily in the formulation of the search problem as defined in Eq. 4. Examples of such design constraints include i) maximum number of PEs allowed in a design for a given application, ii) maximum allowed completion time, iii) additional delays incurred inside a PE (such as those due to internal pipelines in a PE). We illustrate in the next section constraints for such PAs with two-level pipelining.

Example 1 (cont'd). Figure 1 shows a linear PA that minimizes computation time and #PE for multiplying two 4-by-4 matrices using non-pipelined functional units. Note that *computation time* measures the time between the first and last computations in the PA, and does not include load time of inputs and drain time of outputs. For a linear PA, the parameter values are $(t_1, t_2, t_3) = (1, 2, 3)$ and $(k_1, k_2, k_3) = (1, 1, -1)$. Each PE is an inner-product processor, with computation time and $\#PE$ given as

$$\begin{aligned} T_{comp} &= 1 + (N-1)(t_1 + t_2 + t_3) = 19 && (8) \\ \#PE &= 1 + (N-1)(|k_1| + |k_2| + |k_3|) = 10 && (9) \end{aligned}$$

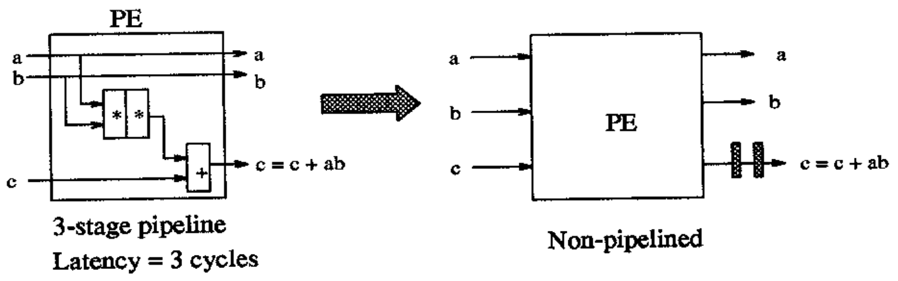

Fig. 2. Pipelining is equivalent to adding delays to dependent data.

In summary, GPM provides for a given application of given size a) a set of parameters for characterizing movements of data in the PA, b) a set of constraints for characterizing correct and desirable operations, c) design objective, and d) an efficient search procedure for finding optimal/feasible designs. It differs from dependency-based approaches proposed by Kuhn, Moldovan, Fortes, and many others in problem representation and traversal of search space.

2.4. New Constraints for Pipelined Functional Units

The effect of pipelining can be summarized by the following simple observation.

Observation 1. *Pipelining with S stages is equivalent to retiming where the delays on all inputs or all outputs, but not both, are increased by S.*

In a PA, two types of data flows exist.

a) *Dependent.* The datum is modified as it flows through the PA; and

b) *Transmittent.* The datum is not modified (read-only).

If the processors have PFUs, only the delays of the dependent data and not the transmittent data have to be increased. Figure 2 shows the effect of pipelining.

The following lemma characterizes the effect of internal pipelines in PEs as new constraints in GPM. We assume that in general each dependent flow i through the PE is computed using a pipeline with S_i stages.

Lemma 1. *Pipelining the functional unit is equivalent to enforcing the following constraints in GPM:*

$$t_i \geq S_i, \quad \textit{for all dependent flows } i, \tag{10}$$

where S_i is the number of stages in the pipeline for dependent flow i.

Proof. An S_i-stage pipeline delays dependent flow i passing through the PE by S_i time steps. Since, by definition, period t_i captures the time difference between two dependent computations of data flow i separated in the index space by vector $\vec{d_i}$, constraint $t_i \geq S_i$ incorporates into GPM the effect of an S_i-stage pipeline to compute the dependent flow i in each PE. □.

Note that pipelining in a PE amounts to a reduction of the search space (as $t_i \geq 1$ for non-pipelined designs). Hence, the computation or completion time (cycle count of the PA) of the optimal two-level pipelined design is more than or at best equal to that of the non-pipelined design. However, the clock rate of a pipelined design can be increased easily to match the same throughput as that of a non-pipelined design. Our approach, therefore, provides a trade-off between the degree of internal pipelining in PFUs and the rate at which PEs are clocked.

2.5. Minimum Initiation Interval Greater Than Unity

The constraints in Eq. 10 assume that two independent operations can be initiated into a PFU in successive cycles without any stage conflicts; that is, MII, the Minimum Initiation Interval, is 1. In general, for pipeline i computing dependent flow i, its reservation table can be in such a way that its $MII = \delta_i > 1$. Since in our model, each PE computes the entire body of the innermost loop in each time step, it is necessary to ensure that no two executions of any PE are within δ time steps of each other, where $\delta = \max_i \delta_i$. The following conditions avoid scheduling two index points within δ steps in any PE.

$$\overset{Min}{\forall \vec{\alpha}} \left|\vec{T}.\vec{\alpha}\right| > \delta, \text{ subject to } \mathbf{K}\vec{\alpha} = 0 \text{ and } \mathbf{D}\vec{\alpha} \neq 0\,, \tag{11}$$

where $\vec{T} = [t_1 t_2 \ldots t_m]$ is the vector of periods, $\mathbf{K} = [\vec{k_1}\vec{k_2}\ldots\vec{k_m}]$ is the matrix of displacements, and $\mathbf{D} = [\vec{d_1}\vec{d_2}\ldots\vec{d_m}]$ is the matrix of dependencies. These conditions say that if two distinct computations are assigned to the same PE, then the time between executing them should be at least δ steps. For each feasible solution (satisfying Eq. 4) found in GPM, Eq. 11 has to be checked, increasing the time complexity to find the optimal parameter values.

3. Pipelined Designs For n-D Meshes

In this section, we present a closed-form condition for deriving pipelined designs from non-pipelined ones for a class of algorithms whose dependence graphs are n-D meshes. For an n-D mesh algorithm, the dependence graph is the identity matrix in n-D (I_n) with $n-1$ transmittent flows of data and one dependent flow. For example, computing a matrix product is a 3-D cube graph algorithm, with **C** as the dependent flow and **A** and **B** as transmitent flows.

Definition 1. *Let $\vec{I} = (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n)^t$ be the index vector of an n-D mesh dependence graph G. We define a cyclic interchange of distance k, denoted as CI_k, as a reindexing of G as follows.*

$$CI_k(G) : (i_{n-k+1}, \cdots, i_n, i_1, \cdots, i_{n-k})^t \longrightarrow (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n)^t. \tag{12}$$

$CI_k(G)$, is a linear transformation of dependence graph G and can be represented by the permutation matrix $\mathbf{P}_k = [\vec{e}_{n-k+1}, \vec{e}_{n-k+2}, \cdots, \vec{e}_n, \vec{e}_1, \vec{e}_2, \cdots, \vec{e}_{n-k}]^t$, where $\vec{e}_i = (\underbrace{0, \cdots, 0}_{i-1}, 1, 0, \cdots, 0)^t$ is the i-th unit vector with a single 1 in the i-th position.

We now extend the definition of cyclic interchange CI_k to apply to PAs as well. A PA for an n-D mesh algorithm has n inputs: $\Omega_1, \Omega_2, \cdots, \Omega_n$. A cyclic interchange, CI_k, for a PA is equivalent to renaming the input variables as follows. $CI_k : (\Omega_{n-k+1}, \cdots, \Omega_n, \Omega_1, \cdots, \Omega_{n-k}) \rightarrow (\Omega_1, \cdots, \Omega_k, \Omega_{k+1}, \cdots, \Omega_n)$ *i.e.*, variable Ω_{n-k+1} is renamed as Ω_1, Ω_{n-k+2} as V_2, and so on. Therefore, when referring to dependence graphs, a cyclic interchange CI_k means a reindexing of the index space. Further, in the context of PAs, a cyclic interchange CI_k means a reindexing of the n-input variables. For example, in a PA for matrix product, a cyclic interchange of distance 1 renames variables (C, A, B) to (A, B, C). Thus, in Figure 4, variable A is replaced by variable C, variable B by A, and variable C by B.

Since PAs for solving a given algorithm are obtained by a linear transformation **T** of dependence graph G, the following lemma shows that PAs for n-D mesh algorithms remain valid under cyclic interchanges of their inputs.

Lemma 2. *Given an n-D mesh algorithm with a dependence graph G, and a PA with inputs* $\Omega_1, \Omega_2, \cdots, \Omega_n$, *the new PA obtained by a cyclic interchange* $CI_k, k = 1, 2, \cdots, n-1$ *of the existing PA is a valid PA that solves the given n-D mesh algorithm.*

Proof. Since mapping a uniform recurrence on a PA corresponds to a linear transformation, cyclically interchanging the inputs of the PA by CI_k is equivalent to reindexing the dependence graph by CI_k. It is easy to see that reindexing by CI_k maps the dependence vectors into each other, resulting in the same set of dependence vectors. Thus, CI_k maps dependence vector $\vec{d_i}$, $i = 1, 2, \cdots, n$, to vector $\vec{d}_{((i-k-1)\mathrm{mod} n+1)}$. By appling the same transformation (**T**) that defines the non-pipelined PA to the cyclically interchanged dependence graph ($CI_k(G)$), we have a new PA whose inputs are a cyclic interchange of inputs in the original PA. In other words, the composite linear transformation $\mathbf{T}\,\mathbf{P}_k$ is also a valid transformation, and PAs remain valid under cyclic interchanges for n-D mesh algorithms, □.

The next lemma establishes the main result for deriving designs of PAs with PFUs from existing non-pipelined ones.

Lemma 3. *Assume an n-D mesh algorithm involving inputs* $\Omega_1, \Omega_2, \cdots, \Omega_n$. *Further, assume that a non-pipelined PA exists for this algorithm with periods* $t_1, \ldots, t_n$, *where* t_1 *is the dependent data flow. To obtain a PA with S-stage pipelined PEs from the original non-pipelined design we can apply* CI_{j-1}, *a cyclic interchange of distance* $j-1$, *if there exists a period* $t_j \geq S$.

Proof. For an n-D mesh algorithm, if $t_1 > S$, then the parameters in GPM correspond to a valid S-stage pipelined PA (Eq. 10). Thus, if there exists some j, $1 \leq j \leq n$, such that $t_j > S$, then by applying CI_{j-1} we obtain a dependence graph whose dependent flow is Ω_j instead of Ω_1. This corresponds to a new PA with its inputs interchanged by the same CI_j. By Lemma 2, cyclically interchanging the inputs results in a valid PA that still solves the same n-D mesh algorithm. □.

Lemma 3 can be used to derive PAs, given existing non-pipelined designs. In the next section, we apply this result as a pruning condition to find optimal PA designs for the matrix-product application.

4. Applications: Matrix Product

Matrix product belongs to an important class of algorithms called 3-D cube graph algorithms. It is fundamental in signal and image processing, and most commercially available DSP processors are evaluated on the number of MAC (Multiply-And-Accumulate) operations they can perform per second. In this section, we present optimal designs of pipelined PAs for computing the product of two $N \times N$ matrices. Since, there is only one dependent flow **C**, let S denote the number of stages in the PFU to compute **C**.

4.1. Evaluation Metric

The total time to compute a matrix product is equal to $T \times t_c$, where T is the number of clock cycles, and t_c is the clock period of each clock cycle. Hence, for a given PA design, its performance can only be improved by increasing its clock rate. This has become increasingly difficult, as there are a fixed number of gate delays (corresponding to the computation of the inner loop body by each PE) that have to be incurred within a clock cycle.

For a given clock rate, it is much easier to design pipelined functional units in each PE and sub-divide the computations (and hence the fixed gate delays) than to design a non-pipelined PE where the entire inner loop has to be executed within a clock tick. As a result, it is simpler to scale the clock rate of a pipelined design than a non-pipelined one. Equivalently, the clock rate of a pipelined design with ALUs of S stages can be up to S-times higher than that of a non-pipelined one. Since it is difficult to predict how many times the clock rate of a pipelined design would be as compared to a non-pipelined one (as it depends whether the designer can split the operation equally and balance the delays), we evaluate an equivalent metric of the ratio of the clock periods as described below.

Let t_c^1 and t_c^S be the clock periods of the non-pipelined PA and a PA with S-stage PFUs (denoted as S-pipelined), respectively. Further, let T_{comp}^1 and T_{comp}^S be the computation times (*i.e.*, clock cycles) of the optimal non-pipelined PA and the optimal S-pipelined PA, respectively. The metric we use for evaluating an S-pipelined design over a non-pipelined design is the ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for both designs to have the same performance; *i.e.*, $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$.

$$t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1 \tag{13}$$

$$\Longrightarrow R = \frac{t_c^S}{t_c^1} = \frac{T_{comp}^1}{T_{comp}^S} \leq 1 \quad (\text{ since } T_{comp}^1 \leq T_{comp}^S). \tag{14}$$

It is easy to see that for the same clock rate, if the pipelined design has the same performance as a non-pipelined one, then all the internal pipelines in the ALUs are fully utilized. Hence, GPM aims to find pipelined designs so that $T_{comp}^S = T_{comp}^1$ or $R = t_c^S = t_c^1 = 1$; *i.e.*, all the stages of the internal pipelines are fully utilized. A value of $R < 1$ implies that the pipelined design has to be clocked faster in order to have the same performance as the non-pipelined one. This also means that some of the stages in the internal pipelines are not being fully utilized.

4.2. Results

Figure 3 shows the 3-D plot of the ratio of clock periods $R = t_c^S = t_c^1$ as a function of the number of pipeline stages (S) and the matrix size (N). The X-Y plane of the 3-D plot also shows the contours for different fixed values of R as S varies from 1 to 25 and the problem size N from 4 to 512 in 8 steps. The N's are plotted in logscale to prevent the compression of smaller values of N. For any given value of N, R stays at the desired maximum value of 1 until S reaches a critical value (S_{crit}) beyond which R starts to fall below 1. For example, for $N = 128$, R is 1 up to $S = 15$; *i.e.*, $S_{crit} = 15$. Similarly, for a given value of S, there is a minimum value of N (N_{crit}) such that GPM can find pipelined designs with $R = 1$. Thus, given a 10-stage PFU (typical of an arithmetic pipeline), GPM is able to find a pipelined PA design for matrix product that can fully exploit the 10-stage pipeline without increasing the cycle count when the matrix size N is at least 64 ($N_{crit} = 64$). These results indicate that for matrix products, given a value

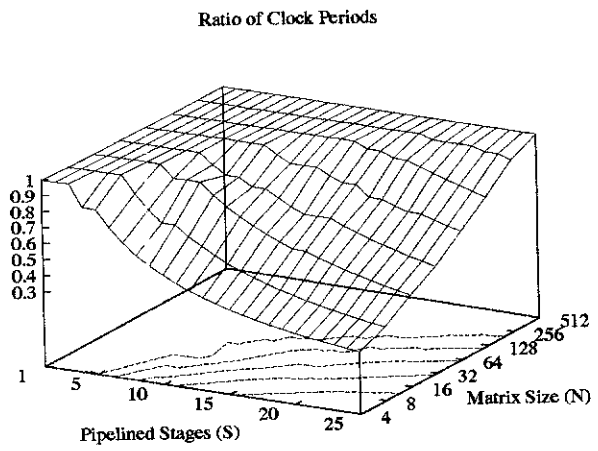

Fig. 3. Ratio of clock periods $R = \frac{t_c^S}{t_c^1}$ in order for the S-pipelined and non-pipelined designs to have the same performance; i.e., $t_c^S \times T_{comp}^S = t_c^1 \times T_{comp}^1$.

of S, there exists a reasonably small matrix size N_{crit}, beyond which the internal pipeline stages can be fully utilized. These results also show the ability of GPM to systematically explore the design space in order to arrive at pipelined designs that optimize the given objective of computation time T_{comp}.

Example 1 (cont'd). In the design presented in Figure 1, input **C** is the only dependent data flow in computing matrix products. Therefore, if we assume 3-stage PFUs for computing inner products, the constraint $t_1 > 3$ is added to account for the PFUs (Eq. 10). Figure 4 shows the data flows for the new PA obtained. The two delays on the C-links in Figure 4 indicate that each PE has a 3-stage pipeline. The different stages of the pipeline in each PE are fully utilized to execute independent operations of 3 C elements. This leads to a computation time of this PA of 19 cycles, which is the same as that of the non-pipelined design ($R = 1$). Comparing the data flows in Figures 4 and 1, we see that the new PA is obtained by replacing the C input by B, B by A, and A by C.

Lemma 3 has been applied in generating the results in Figure 3. For instance, given a matrix size $N = 64$, the optimal non-pipelined design has periods $(t_1, t_2, t_3) = (4, 5, 10)$, resulting in $T_{comp} = 1198$. Using Lemma 3, the optimal pipelined designs for S varying from 1 to 10 can be obtained by CI_2 (interchange of distance 2) without any search, as the objective function T_{comp} (which is the sum of the three periods) evaluates to the same value under any cyclic interchange of inputs. From the above parameter values and Lemma 3, it is easy to see that all the optimal designs for $S = 1, \cdots, 10$ have the same computation time of 1198 cycles; *i.e.*, $R = 1$ for $S = 1, \cdots, 10$. Consequently, Lemma 3 can be used as an effective pruning condition in GPM to define the flat portion of the surface in Figure 3, and can be used in trade-off analysis discussed in the next section.

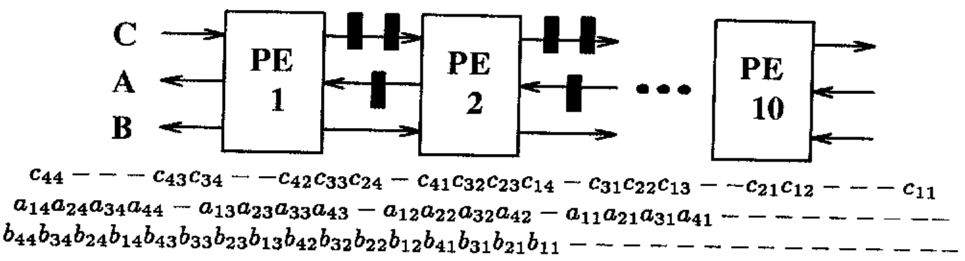
C
A
B
PE 1
PE 2
PE 10
$c_{44} - - - c_{43}c_{34} - -c_{42}c_{33}c_{24} - c_{41}c_{32}c_{23}c_{14} - c_{31}c_{22}c_{13} - -c_{21}c_{12} - - - c_{11}$
$a_{14}a_{24}a_{34}a_{44} - a_{13}a_{23}a_{33}a_{43} - a_{12}a_{22}a_{32}a_{42} - a_{11}a_{21}a_{31}a_{41} - - - - - - - - -$
$b_{44}b_{34}b_{24}b_{14}b_{43}b_{33}b_{23}b_{13}b_{42}b_{32}b_{22}b_{12}b_{41}b_{31}b_{21}b_{11} - - - - - - - - - - - - - - - -$

Fig. 4. A PA for matrix multiplication with 3-stage PFUs.

Note that this is obtained by a cyclic interchange of the inputs as compared to those shown in Figure 1.

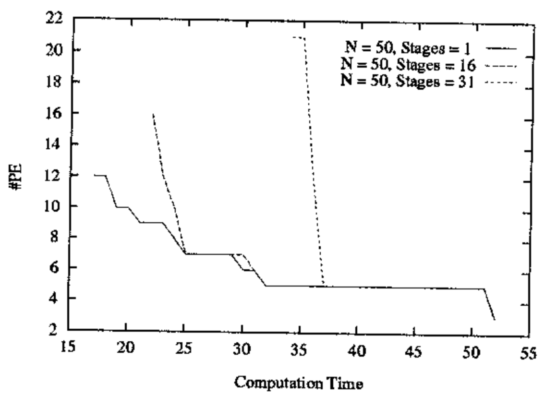

Fig. 5. Performance trade-offs: Variation in $\#PE$ with T_{comp}.

4.3. Processor-Time Trade-offs

GPM can be used to generate optimal designs when there are constraints on processor count $\#PE$ or computation time T_{comp} for a given number of stages S in each PFU. Figure 5 shows the trade-offs between $\#PE$ and T_{comp} for computing 50×50 matrix products and three values of S. A particular trade-off curve (with a fixed value of S) shows the minimum $\#PE$ (*resp.*, T_{comp}) required, given an upper bound on the computation time T^{ub}_{comp} (*resp.*, $\#PE^{ub}$). For example, given a bound $T_{comp} \leq T^{ub}_{comp} = 25$, Figure 5 shows that the minimum $\#PE$ needed is 7, if each PE has a PFU with at most 16 stages. Thus, the designer can use this figure to arrive at a final design, given restrictions on resources such as $\#PE$ or T_{comp}. Figure 5 also shows that, given a fixed number of PEs, the minimum computation time required increases as the number of stages in the PFU increases. This is true because it gets harder to utilize the PFUs efficiently. For instance, if $\#PE \leq 12$, then $T^{1}_{comp} \geq 17$, $T^{16}_{comp} \geq 23$, and $T^{31}_{comp} \geq 36$. Further, the plot shows an increasing initial steep decline as S is increased. Hence, for larger S, there are more attractive alternatives than the time-optimal or processor-optimal designs.

5. Comparisons With Existing Work

There have earlier efforts by Kung and Lam [3] and recently by Valero-Garcia *et el.*, [4] to obtain two-level pipelined PAs. They used a common approach that retimes a PA in order to include additional delays for pipelining. Their approach, however, is restricted to PAs that have already been designed; that is, they took an existing PA and derived a two-level pipelined one by retiming. Since pure addition of delays only increases the clock-cycle count of the retimed PA, they coalesced the PEs in order to obtain better utilization and improved execution time. Valero-Garcia, *et al.* [4], further showed that linear PAs with contra-flow data can be retimed and the PEs coalesced in order to obtain the same clock-cycle count.

The drawback with these previous approaches is that coalescing is not always possible. In fact, for lower dimensional PAs, there is no systematic technique to coalesce PEs by a factor of S. In addition, these methods cannot systematically explore the space of pipelined designs because they start with a particular PA and are restricted only to the data flows defined in it. For instance, the PA shown in Figure 4 cannot be obtained by retiming and coalescing, as there is a new data-flow pattern that is a cyclic interchange of the old one. In contrast, GPM can be used as a systematic tool to explore the entire space of pipelined PAs, and arrive at a design that optimizes a user-specified objective and satisfies design constraints.

6. Conclusions

In this paper, we have extended GPM, the General Parameter Method, for designing two-level pipelined processor arrays. Our method can explore the space of feasible PAs to obtain a PA that optimizes a given objective. In contrast, existing methods can only retime a given design for two-level pipelining; hence, they cannot be used to find other possibly better designs that are not obtained by retiming. We present a new design for matrix multiplication that cannot be obtained by existing methods, and show the tradeoffs (in Figure 3) between the degree of internal pipelining and the rate at which PEs are clocked. Our new design requires a new data-input pattern that is obtained by a cyclic interchange of the old one. Finally, we give closed-form results for an important class of algorithms represented by n-D meshes, which can be used as constraints in GPM to reduce the search complexity.

References

1. K. N. Ganapathy and B. W. Wah, Synthesizing optimal lower dimensional processor arrays, *Proc. Int. Conf. on Parallel Processing* (Pennsylvania State University Press, 1992).
2. K. N. Ganapathy and B. W. Wah, Optimal design of lower dimensional processor arrays for uniform recurrences, *Proc. Application Specific Array Processors* (IEEE Computer Society Press, 1992) 636–648.
3. H. T. Kung and M. Lam, Wafer-Scale integration and two-level pipelined implementations of systolic arrays, *J. Parallel and Distributed Computing* 1 (1984) 32–63.
4. M. Valero-Garcia, J. J. Navarro, J. M. Llaberia, and M. Valero, Systematic hardware adaptation of systolic algorithms, *Int. Symp. on Computer Architecture* (1989) 96–104.