Designing a Scalable Processor Array for Recurrent Computations

Kumar N. Ganapathy, Benjamin W. Wah, *Fellow*, *IEEE*, and Chien-Wei Li

Abstract—In this paper, we study the design of a coprocessor (CoP) to execute efficiently recursive algorithms with uniform dependencies. Our design is based on two objectives: 1) fixed bandwidth to main memory (MM) and 2) scalability to higher performance without increasing MM bandwidth. Our CoP has an access unit (AU) organized as multiple queues, a processor array (PA) with regularly connected processing elements (PEs), and input/output networks for data routing. Our design is unique because it addresses input/output bottleneck and scalability, two of the most important issues in integrating processor arrays in current systems. To allow processor arrays to be widely usable, they must be scalable to high performance with little or no impact on the supporting memory system. The use of multiple queues in AU also eliminates the use of explicit data addresses, thereby simplifying the design of the control program. We present a mapping algorithm that partitions a data dependence graph (DG) of an application into regular blocks, sequences the blocks through AU, and schedules the execution of the blocks, one at a time, on PA. We show that our mapping procedure minimizes the amount of communication between blocks in the partitioned DG, and sequences the blocks through AU to reduce the communication between AU and MM. Using the matrix-product and transitive-closure applications, we study design trade-offs involving 1) division of a fixed chip area between PA and AU, and 2) improvements in speedup with respect to increases in chip area. Our results show, for a fixed chip area, 1) that there is little degradation in throughput in using a linear PA as compared to a PA organized as a square mesh, and 2) that the design is not sensitive to the division of chip area between PA and AU. We further show that, for a fixed throughput, there is an inverse square root relationship between speedup and total chip area. Our study demonstrates the feasibility of a low-cost, memory bandwidth-limited, and scalable coprocessor system for evaluating recurrent algorithms with uniform dependencies.

Index Terms—Access unit, affine dependencies, area index, clock-rate reduction, dependence graph, memory bandwidth, multimesh graph, partitioning, processor array, scheduling, uniform dependencies.

1 Introduction

PROCESSOR arrays have been designed for efficient computation of recurrences. In mass quantities, the production of such fixed-function arrays is manageable and economical. However, when a single processor array is to be used for a new application, a manufacturer will have to take the long and costly process of designing and fabricating the application-specific integrated chip. Although the cost of such designs has decreased in recent years, budget constraints have motivated a trend away from custom hardware development except in cases in which the performance required justifies the cost of developing such specialized hardware. Consequently, general-purpose or programmable processor-array architectures are more attractive alternatives.

This paper discusses the design of a parallel VLSI coprocessor (CoP) that is programmable for computing recurrences. The CoP interfaces with a front-end host machine that is responsible for data input and control. The overall requirements on the CoP are:

- Fixed data bandwidth to main memory for easy integration into existing systems;
- Scalability, or ability to increase performance by adding processor/memory modules without increasing the bandwidth to main memory; and
- Ability to execute efficiently the class of algorithms with uniform dependencies.

The last requirement indicates the domain of applications targeted for CoPs. These include nested-loop algorithms with uniform dependencies, which involve uniform recurrences (UREs), and "uniformized" affine recurrences (Section 3). Such loops are found frequently in signal and image processing, scientific computations, matrix and linear algebra computations, optimization, digital communications, and control. Although an application-specific design for each application would result in higher performance, we have chosen a common retargetable architecture that can be reused for a number of applications. The proposed architecture can be visualized as a coprocessor to workstations or as a VLSI pipeline in supercomputers for loop computations, similar to vector functional units for executing vector instructions.

Our approach is to design a combination of hardware architectures and software mapping methods (Section 4) under fixed main-memory bandwidth (Section 2.1). The trade-offs we consider include the following (Section 5).

- For a given throughput, reduce clock frequency by increasing area;
- For a given clock rate, increase throughput by increasing area.

- *K.N. Ganapathy is with the Telecommunications Division, Rockwell International, Newport Beach, CA 92658. E-mail: kumar@nb.rockwell.com.*
- *B.W. Wah and C.-W. Li are with the Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, Urbana, IL 61801. E-mail: {wah, cwli}@manip.crhc.uiuc.edu*

Manuscript received 9 Aug. 1994; revised July 1996.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number 100936.1.

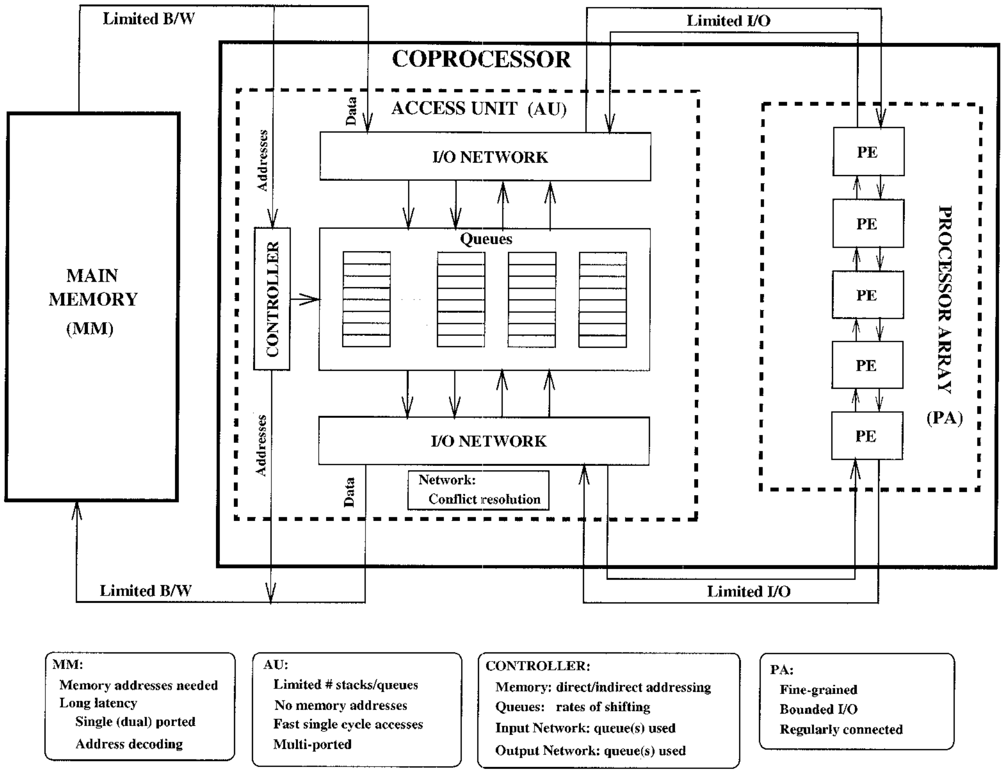

Limited B/W
Limited I/O
COPROCESSOR
ACCESS UNIT (AU)
Data
Addresses
I/O NETWORK
Queues
CONTROLLER
MAIN MEMORY (MM)
I/O NETWORK
Network: Conflict resolution
PE
PROCESSOR ARRAY (PA)
Limited B/W
Limited I/O
MM: Memory addresses needed; Long latency; Single (dual) ported; Address decoding
AU: Limited # stacks/queues; No memory addresses; Fast single cycle accesses; Multi-ported
CONTROLLER: Memory: direct/indirect addressing; Queues: rates of shifting; Input Network: queue(s) used; Output Network: queue(s) used
PA: Fine-grained; Bounded I/O; Regularly connected

Fig. 1. Coprocessor architecture proposed to solve a class of algorithms modeled by uniform recurrences.

There have been numerous efforts to develop general-purpose systolic computers in the past ten years. These include Warp and iWarp [1], Matrix-1 [8], SLAPP [6], medium-grain architecture for image and signal processing [27], VATA [30], pseudosystolic linear array [22], [24], and a host of others. However, many of these designs have powerful processors with large local memories, and high-bandwidth data interconnect between processors and the host/global memory. Hence, the cost of such systems is very high. In our approach, we aim at designing a simple, resource-limited VLSI array processor that can be attached to any standard single-port main memory (with fixed bandwidth and long latencies) that results in a system with low cost and acceptable performance for our target applications. This is in contrast to most existing architectures in which memory bandwidth is increased proportionally as the system is scaled for higher performance.

Recently, there have been efforts to develop a systematic approach to partition and map matrix algorithms on mesh-connected processor arrays [24], [23], [20]. Our approach differs in trying to extract maximum data re-use under the constraint of a low-bandwidth interconnect to main memory in our CoP. This research also differs from traditional work on mapping/partitioning data on fixed-size processor arrays [21], [19], [15], [2], [25], [29], [17], [5] by assuming only limited storage space in the processor array and by considering the effect of main-memory latency due to low-bandwidth interconnection to the main memory.

The rest of this paper is organized as follows. We first present our proposed array processor (Section 2), describe the target algorithms (Section 3) and the mapping and partitioning techniques (Section 4), evaluate and discuss results using matrix product and transitive closure as example applications (Section 5), and discuss the impact of clock-rate reduction on overall system performance (Section 6).

2 Coprocessor Architecture and Rationale

The architecture studied in this paper consists of the following components (Fig. 1):

- An external main memory (MM) for storing input and output data,
- A dedicated processor array (PA) for executing the computations of a given algorithm, and
- An access-unit (AU) for streamlining the flow of data between PA and MM.

Section 2.1 describes the individual components in more detail. Section 2.2 compares our architecture to existing architectures.

2.1 Coprocessor Architecture

Main Memory: (MM) The main memory in the architecture is a standard (usually interleaved) memory for storing data involved in the computations. The data in MM are accessed by supplying a stream of memory addresses to the address decoder in MM. The bandwidth of accesses is usually constrained by a small number of memory ports, and additional latency is incurred in decoding addresses. The assumption of accessing data from standard MM with low bandwidth simplifies the integration of CoPs to conventional hosts.

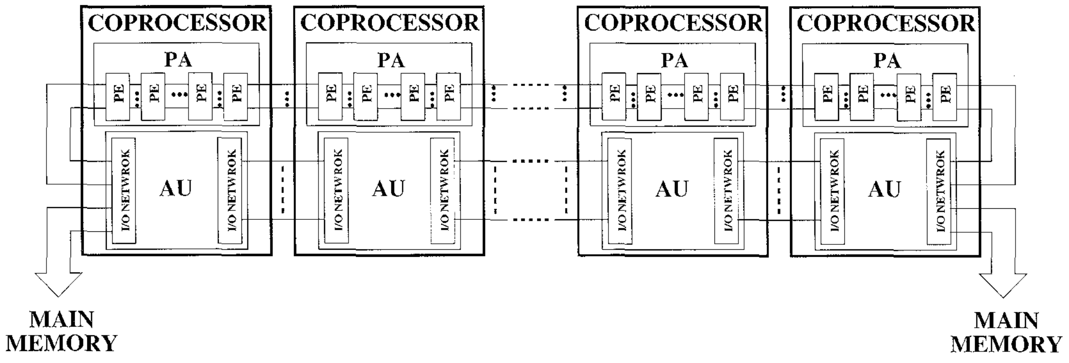


Fig. 2. Concatenation of CoPs to solve larger problems or to improve speedup.

Note that we have abstracted the interface between CoP and MM as a port whose performance is aggregated by its bandwidth. In implementing the CoP architecture, the specific memory interface, such as Rambus, and its handshaking must be considered and be designed in hardware.

Access-Unit: (AU) To overcome the bandwidth limitation and to mask the long memory latency of a shared MM, we include in our architecture a buffer memory called access-unit to store data accessed from MM and to feed PA at a much higher rate. AU has a fixed amount of storage to buffer the intermediate data that cannot be held in PA so that these data can be recirculated to PA to reduce the demand on MM.

The storage in AU is organized as FIFO queues, and explicit memory addresses (except for queue numbers) are not used. In each cycle, the data present at the head of some of the queues are sent to PA or to MM through the output network, and data from PA or MM are sent to the tail of the queues. In addition, AU can

1) prefetch data from MM into its queues to hide the memory latency,
2) shift the different queues at different rates to reorder the data relative to each other,
3) perform indirect addressing of MM in which a sequence of addresses obtained from MM is subsequently used to access data in MM.

This indirect addressing corresponds to scatter-gather instructions in vector computes.

The use of queues in AU saves address bits, simplifies address decoding, and reduces silicon area in implementation. Further, the queue structure permits AU size to be scaled without changing the number of address bits, number of ports to MM, and number of ports to PA.

Processor Array: (PA) The processor array is a regularly connected array of processing elements (PEs). Each PE in PA has a microprogrammed control that governs its operations. The control specifies the actions to be performed in a PE on receiving data from its neighbors. For example, to solve a matrix product expressed as a 3D recurrence, all PEs perform an inner-product computation $c = c + a \cdot b$, where c, a, b are elements of matrices **C**, **A**, and **B**, respectively.

The choice of the PE configuration is dictated by the requirement of scalability. For true hardware scalability, the number of memory ports, number of ports to PA, and number of address bits should be independent of the size of the architecture and the size of the problem being solved. This implies that PA should be I/O bounded with a constant number of ports. A possible configuration, then, is a linear array of PEs with two boundary PEs that communicate with the heads of the queues in AU. (A 2D mesh configuration is less desirable as the number of peripheral PEs is proportional to the square root of the total number of PEs.) A linear array of PEs generally has lower throughput than a square array with the same number of PEs for executing nested loops with uniform dependencies [9], [11], [7], [12]. However, for the algorithms with multimesh dependence graph (DG) (see Section 3) and when MM bandwidth is fixed, we show in Section 5.4 that this degradation is small and that speedup is bounded by the times to read input matrices and to write output matrices.

Using a linear PA configuration, Fig. 2 shows a concatenation of several CoPs that can be used to improve the performance of solving a given problem or to solve a problem of larger size. The FIFO queues of AU can be split over multiple chips when the entire buffer memory required for a large number of PEs cannot fit in a single chip. This results in a linear array of PEs with a linear array of AUs to buffer data. While each individual CoP can have an optimal division of chip area between PA and AU (as shown in Section 5.3) to fully mask the MM latency, the linear concatenation of CoPs may result in suboptimal division of the total silicon area across multiple chips and lower performance. However, we show in Section 5.4 that the performance of a linear PA is rather insensitive to AU size, resulting in a modular expansion of the system without significant degradation in performance.

For a given PA configuration, an important issue to be studied is the allocation of a fixed chip area between PA and AU. This issue, to be studied in Section 5.3, has not been addressed in other architectures based on memory queues.

In short, our architecture is characterized by the following parameters:

- #PE: number of PEs,
- b: local Memory in each PE,
- B_{mm}: ratio of bandwidths between AU-MM and AU-PA,
- p: size of AU in blocks,

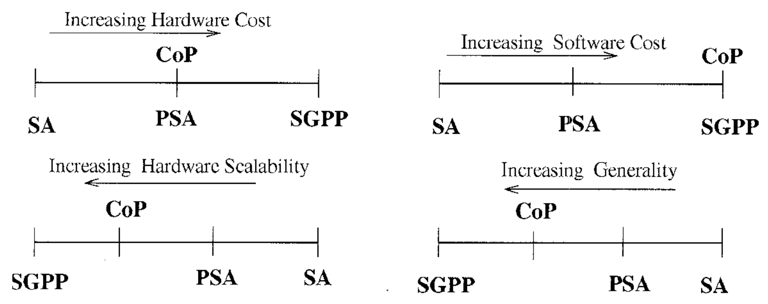


Fig. 3. Relative comparison of CoP to other architectures with limited memory bandwidth.

- A_{pe}: chip area for implementing a PE in unit of memory words (A_{pe} of 100 means that each PE occupies an area equivalent to 100 words of memory),
- Topology of the PEs: linear or square.

Our procedure for designing CoPs has two major steps:

1) define a model of the architecture in terms of the parameters defined above, and
2) arrive at the final architecture by analyzing cost and performance trade-offs.

2.2 Comparisons to Existing Architectures

In this section, we perform a qualitative comparison of our CoP to other existing architectures with limited MM bandwidth. The attributes used are:

1) hardware cost,
2) software cost,
3) generality, and
4) hardware scalability.

The last attribute measures the ability to increase performance by adding more processing units to the system in a modular fashion.

The architectures considered in our comparison are:

- Systolic "General-Purpose" Processors (SGPP). These are programmable general-purpose PAs that have been built for a class of applications. Examples include iWarp (in systolic mode) [1], SLAPP [6], Matrix-1 [8], and medium-grain image processing architectures [27].
- Partitioned Systolic Arrays (PSA). These include research efforts aimed at designing fixed-size systolic arrays for solving large problems [21], [19], [15], [2], [25], [29], [17], [3], [5], [32].
- Systolic Arrays (SA). These refer to traditional, algorithm-specific, problem-size-dependent systolic arrays.
- Coprocessor (CoP). Our proposed coprocessor.

We do not compare our CoP to commercially available shared-memory multiprocessors (SMM) and distributed-memory multicomputers (DMM), as these systems have their memory bandwidth tuned to a specific configuration. An SMM is attached through a dedicated interconnection network to a set of memory modules. As an SMM is scaled to higher performance, the number of memory modules and the size of the network must also be increased. Likewise, a DMM with local memory in each processor also has increased memory sizes and bandwidths, as the system is scaled to higher performance.

Fig. 3 compares the different architectures on the chosen attributes. Note that the figure only shows the relative ordering between different architectures and does not represent the actual performance difference.

In terms of hardware cost (or, equivalently, hardware complexity), SA is the simplest with SGPP being the most expensive. CoP has about the same hardware complexity as PSA with an external buffer for recirculating data.

In terms of software cost or programmability, SA is the lowest, as there is no programming effort once the hardware is designed. Programmability of CoP is similar to that of SGPP: Both accept high-level sequential programs as input and use a compiler to map the executions.

For generality, SA is the most restricted in terms of the application and the problem size it is designed for. PSA relaxes the restriction on application size but is still tied to an application. CoP is more general than PSA because multiple applications can be mapped on the same hardware architecture. SGPP is the most general as it can solve similar application problems (modeled as UREs) as CoP, as well as problems solvable on a general purpose DMM.

In terms of hardware scalability, SA is problem-size dependent and, hence, not scalable. PSA may be scalable if its architecture is designed properly. CoP and SGPP are both scalable by adding new PEs and local memory (AU in the case of CoP) to an existing system.

To summarize, our proposed CoP has low hardware cost, is programmable to solve a class of application problems, is scalable to a larger system by connecting additional PEs and AUs, and relies on a compiler to map computations in order to effectively exploit parallelism. Its design represents a good trade-off among the four attributes to solve the class of application problems considered in this paper. In Sections 4 and 5, we present methods to find the optimal partitioning of chip area between PA and AU.

3 Target Algorithms and Transformations

In this section, we define the set of applications that can be handled by our proposed CoPs. The application domain is the set of nested-loop algorithms that can be modeled as multimesh graphs (MMGs) [22], [23]. These can be used to represent the class of uniform dependencies, and, if the recurrence is affine but not uniform, to uniformize the dependencies. Our starting points in our mapping process described in Section 4 are, therefore, *multimesh graphs of (uniformized) UREs.*

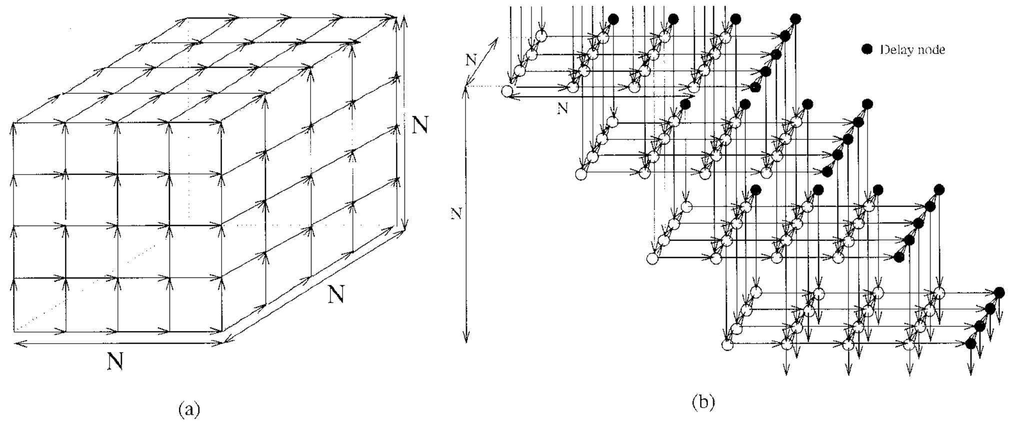


Fig. 4. 3D MMGs after applying Moreno and Lang's regularization procedure [22], [23]. (a) Product of two N-by-N matrices. The MMG is an $N \times N \times N$ cube with unit vectors along the axes as dependence vectors. (b) MMG for the transitive closure of a 4-by-4 matrix.

For instance, the following Fortran-like nested loops can be represented by a system of recurrence equations.

```
DO $(j_1 = l_1, u_1 \; ; \; j_2 = l_2, u_2 \; ; \cdots ; \; j_n = l_n, u_n)$
    $S_1(\vec{J})$ ;
    ⋮
    $S_b(\vec{J})$ ;
END
```

The column vector $\vec{J} = [j_1, j_2, \ldots, j_n]^T$ is the index vector (or index point). $S_u(\vec{J})$, $u = 1, \ldots, t$, are t assignment statements in iteration $\vec{J}$ having the following form:

$$Z_i\left(y(\vec{J})\right) = \phi\left[Z_1\left(x_1(\vec{J})\right), \ldots, Z_r\left(x_r(\vec{J})\right)\right], \tag{1}$$

where $1 \le i \le r$.

Nested-loop structures have a direct correspondence with recurrence equations that provide a succinct mathematical representation for them. An n-dimensional recurrence equation is equivalent to a set of n nested loops, whose loop-carried dependencies correspond to dependencies in the recurrence equation.

Recurrences can be classified as uniform or nonuniform based on the nature of their dependencies [16]. A recurrence equation, $Z(\vec{p}) = \phi[Z(\vec{q}_1), Z(\vec{q}_2), \ldots, X(\vec{q}_r)]$, is called *uniform* (URE) if $\vec{q}_i = \vec{p} + \vec{d}_i$, where $\vec{d}_i$ is a constant n-dimensional vector independent of $\vec{p}$ and $\vec{q}_i$. A recurrence equation is called *affine* or *linear* (LRE) if $\vec{q}_i = \mathbf{A}_i\vec{p} + \vec{b}_i$, where $\mathbf{A}_i$ is a constant-coefficient $n \times n$ matrix, and $\vec{b}$ is an n-dimensional vector. A recurrence equation is called *nonlinear* if $\vec{q}_i = \chi(\vec{p})$, where χ is a nonlinear function.

In the remainder of this paper, we use the dependence graph (DG) of a nested loop as a graphical tool to describe the partitioning and mapping procedures. The DG of an n-nested loop algorithm is defined over an n-dimensional integer lattice domain, where nodes correspond to operations inside the nested loops, and arcs correspond to loop-carried dependencies.

In this work, we restrict ourselves to recurrences with uniform dependencies, which involve uniform recurrences and "uniformized" affine recurrences. Hence, only structural information of the algorithm, i.e., index set and dependence matrix, is needed. The following examples show the recurrences and nested-loop representations of the matrix-product and transitive-closure algorithms.

EXAMPLE 1. The following nested-loop program is the uniformized version of the matrix-multiplication algorithm.

```
DO $(i = 1, N \; ; \; j = 1, N \; ; \; k = 1, N)$
    $A(i, j, k) = A(i, j-1, k)$
    $B(i, j, k) = B(i-1, j, k)$
    $C(i, j, k) = C(i, j, k-1) + A(i, j, k) \times B(i, j, k)$
END
```

Intuitively, data $A_{i,k}$ are pipelined along the j axis, and $B_{k,j}$, along the i axis.

EXAMPLE 2. Consider the following three-dimensional (3D) recurrence with $n = 3$, $r = 5$:

$$Z(k, i, j) = \mathbf{X}(k, i)\mathbf{Y}(j, k) + Z(k-1, i+1, j+1) + Z(k-1, i+1, j) + Z(k-1, i, j+1). \tag{2}$$

After pipelining and uniformization, (2) becomes

$$Z(k, i, j) = X(k, i, j-1)\, Y(k, i-1, j) + Z(k-1, i+1, j+1) + Z(k-1, i+1, j) + Z(k-1, i, j+1). \tag{3}$$

The resulting set of dependencies are equivalent to those of Floyd-Warshall algorithm to compute the transitive closure of an $N \times N$ matrix.

Before mapping a recurrence on a CoP, we transform its DG using Moreno and Lang's uniformization or regularization procedure [22], [23] to convert the DG to a regular multidimensional mesh graph called *multimesh graph* (MMG). Although this procedure may introduce unnecessary operations in processing the recurrence, it allows the partitioning algorithm to partition the MMG into blocks of uniform sizes and shapes, and simplifies the mapping of these blocks in AU and their scheduling on PA (which in turn reduces the amount of control information needed in CoP).

The regularization procedure converts a fully parallel DG of a given matrix algorithm to a regular MMG by per-forming transformations to remove broadcasting, bidirectional flow of data, and irregular dependencies. Informally, a matrix algorithm is described recursively by an outer loop with a loop body of vector, scalar, and other matrix algorithms.[1] The restriction is that each node of the DG can have at most three operands. This property allows transformations to regularize the DG to an MMG with a 3D cube structure. Fig. 4a shows the cubical-mesh DG for the matrix-product algorithm in Example 1.

For the transitive-closure problem described in Example 2, we use Moreno and Lang's regularization procedure [22] to obtain a 3D MMG with a cubical structure (Fig. 4b). The darkened nodes in the figure are delay nodes, which are added in the regularization procedure to obtain a cubical structure. We have chosen such a 3D MMG with delay nodes as it presents a uniform and simple method of executing the MMG. For an algorithm-specific design, the delay nodes would contribute to a significant portion of the execution time; hence, the MMG structure was not used in the design of systolic processors for computing transitive closures. For a CoP with a low bandwidth port to an external MM, the regular 3D MMG structure with additional delay nodes is justified as it simplifies the mapping (using fewer control bits) and access patterns to MM. Hence, in studying the transitive closure problem in this paper, we map the corresponding 3D MMG to our CoP.

We have chosen not to start from the recurrence definition of an application problem and derive its corresponding MMG before mapping it to a CoP. It is not necessary to show such a step because the technique of deriving MMGs is well understood in the literature [22], [23]. Although one MMG may be better than another when mapped to a CoP, our focus in this paper is to illustrate how our mapping technique can be applied to map MMGs. As this process can be automated, users can apply it in conjunction with the process of deriving MMGs when given a new application problem in the future. A secondary reason for not showing the derivation of MMGs is due to space limitation.

4 Mapping Process

In this section, we describe our method of mapping the high-level loop specification of an algorithm specified as MMGs to our target CoP. The entire mapping process, depicted in Fig. 5, can be broken down into the following steps:

1) Partition the MMG into blocks, where data in a block can be processed by PA in one pass without further partitioning (Section 4.1).
2) Schedule the execution of a single block on PA using GPM (Generalized Parameter Method, Section 4.2). This step is to get an optimal design of PA, in terms of computation time or other objectives and under various constraints. The design parameters determined in this step are the number of PEs in PA, local memory in each PE, and I/O bandwidth of PA.
3) Sequence the blocks through AU; i.e., determine which blocks will occupy AU at each time step (Section 4.3). The goal of this step is to establish a balanced division of chip area between PA and AU in order to minimize the penalty of MM accesses. In this step, AU size is fixed.
4) Generate address and code sequences from Steps 2 and 3. This step, although well-defined and well understood, is not trivial in implementation. The development of the code generator is ongoing at this time.

1. For a formal definition, see Chapter 5 of [23].

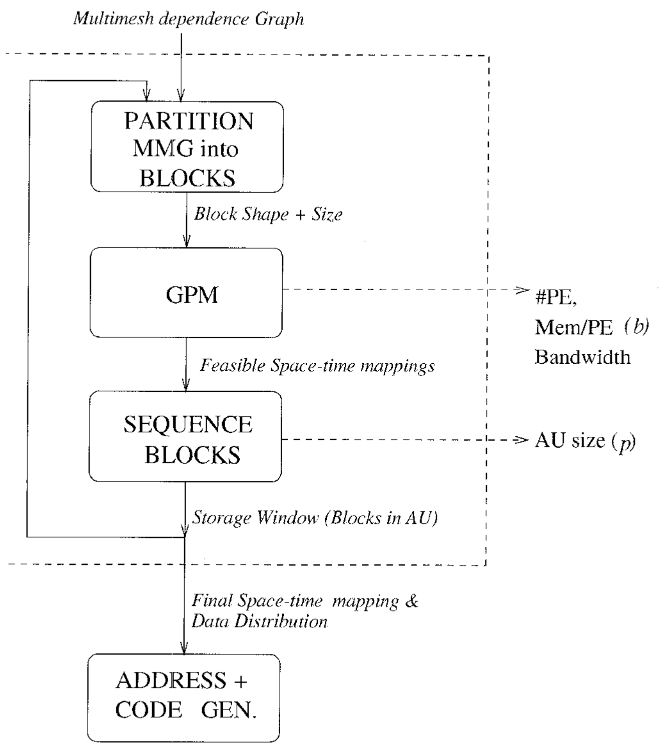

Multimesh dependence Graph
PARTITION MMG into BLOCKS
Block Shape + Size
GPM
#PE, Mem/PE (b) Bandwidth
Feasible Space-time mappings
SEQUENCE BLOCKS
AU size (p)
Storage Window (Blocks in AU)
Final Space-time mapping & Data Distribution
ADDRESS + CODE GEN.

Fig. 5. Mapping the MMG of an algorithm to a CoP architecture.

The last step is well-defined once partitioning, sequencing, and scheduling are done. Hence, in the rest of this section, we describe only the first three steps of the mapping process.

In addition to the symbols defined in Section 2.1, we define the following symbols:

- N: size of each dimension of a given MMG,
- m: size of each dimension of a parallelepiped (block) after partitioning (for the matrix-product and transitive-closure problems, the blocks are cubical of size $m \times m \times m$),
- n: number of dimensions of the recurrence equation, and
- p: size of AU in unit of blocks (where the p blocks are arranged in a $\sqrt{p} \times \sqrt{p}$ tile).

4.1 Partitioning an MMG

The objective in this step of the mapping process is to partition a given MMG into nonoverlapping blocks of a maximum size that can be processed by PA in one pass. This step is necessary because PA has only a limited number of PEs and I/O ports.

In our partitioning procedure, we first define a partitioning matrix whose columns represent hyperplanes that partition an MMG into blocks. We then present our partitioning procedure and demonstrate its properties. (For brevity, proofs are presented in Appendix A.) Finally, we illustrate the procedure by two examples.

Methods for finding independent partitions, in which the communication between blocks is zero, have been proposed before [26], [28]. However, when the original DG has only one connected component (for algorithms considered in this paper), independent partitioning results in only one block, i.e., the entire DG. A technique called supernode partitioning has been proposed [14] to partition nodes in a DG that depend on each other, and reduce communication between supernodes by propagating results inside a supernode. However, a systematic way to find such partitions is not presented.

Partitioning algorithms have also been studied in the context of DMMs. Here, the goal is to partition data in order to maximize parallelism and to reduce data communication among processors. Since the goal of our partitioning algorithm is different and aims to minimize the amount of data transferred between AU and MM, data partitioning algorithms developed for DMMs can only be used as heuristics in CoPs. In other cases, DMM partitioning schemes have been developed for more restricted cases and are not applicable here. For instance, Kulkarni et al. [18] proposed a unimodular loop transformation for partitioning an iteration space in order to increase parallelism, achieve load balance, and minimize interpartition communications. However, their scheme cannot be applied in CoPs because it can only partition dependence graphs in 2D iteration space or doubly nested loops along one direction. It is desirable to have a general scheme that can handle higher dimensional nested loops and can partition along multiple directions in the iteration space.

Our approach to partitioning in this paper is similar to that of Moldovan and Fortes [21]. For an n-D algorithm, we find n-independent hyperplanes to partition a DG into blocks. Hence, our blocks are "parallelepipeds," and their shape can be described by a partitioning matrix $\mathbf{P}$ consisting of n partitioning vectors.

$$\mathbf{P} = \left[\vec{p}_1\ \vec{p}_2\ \vec{p}_3\ \cdots\ \vec{p}_n\right]. \qquad (4)$$

Since we are dealing with uniform dependence algorithms, we restrict the sizes and shapes of all the blocks to be the same. This simplifies the address and code generation for PEs. The reason for choosing exactly n partitioning vectors in an n-D domain is that blocks will not be regular if we have other than n partitioning hyperplanes.

EXAMPLE 3. In Fig. 6, the index set is a 2D plane, and the blocks formed by the hyperplanes are normal to vectors $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$. Choosing $\vec{p}_1$ or $\vec{p}_2$ alone results in unequal or unbounded blocks. Similarly, choosing $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$ results in unequal blocks. Choosing $\vec{p}_1$ and $\vec{p}_2$ forms blocks of equal sizes except at the boundaries.

DEFINITION 1. *Given an n-D DG G of an algorithm and a partitioning matrix* $\mathbf{P}$*, G_b, the block-level DG of G, is a DG in n-D space, where*

- *Nodes in G_b correspond to blocks of G which consists of all of the nodes within the "parallelepiped" defined by the n partitioning hyperplanes in* $\mathbf{P}$*, and*
- *Edges in G_b correspond to dependence vectors crossing the hyperplanes between adjacent blocks in G.*

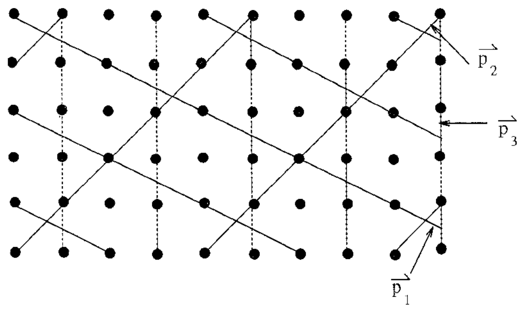


Fig. 6. Hyperplane partitioning of dependence graphs. The index set is a 2D plane. The vectors $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$ correspond to a family of regularly spaced hyperplanes normal to the index set. They determine only the orientation of the hyperplanes, i.e., block shape, and not the spacing among them.

The following lemma presents the conditions for choosing a valid partitioning matrix $\mathbf{P}$. (The proof is shown in Appendix A.1.)

LEMMA 1. *The partitioning of a DG by partitioning matrix* $\mathbf{P}$ *is valid if and only if* $\mathbf{P}^t\mathbf{D} \geq \vec{0}$ *or* $\mathbf{P}^t\mathbf{D} \leq \vec{0}$*, where* $\mathbf{D}$ *is the dependency matrix.*

Next, we present a procedure for choosing a good partitioning vector that results in a very small amount of communication between adjacent dependent blocks.

PROCEDURE 1. Let $g = rank(\mathbf{D})$, where $\mathbf{D} = [\vec{d}_1\vec{d}_2\ldots\vec{d}_k]$ is the dependency matrix. Hence, only g of the k dependence vectors are linearly independent. Without loss of generality, assume that the first g columns are linearly independent, and let $\mathbf{D}' = [\vec{d}_1\vec{d}_2\ldots\vec{d}_g]$ be an $n \times g$ matrix consisting of the g linearly independent vectors of $\mathbf{D}$. Let $\mathbf{D}'_i$ be an $n \times g - 1$ matrix derived from $\mathbf{D}'$ by dropping the ith column vector; i.e., $\mathbf{D}'_i = [d_1\ldots d_{i-1}d_{i+1}\ldots d_g]$. The number of partitioning hyperplanes needed to partition the DG is g (since $g = rank(\mathbf{D})$). Hence, matrix $\mathbf{P} = [\vec{p}_1\ldots\vec{p}_g]$ is chosen such that $\vec{p}_i$ is given by

$$\vec{p}_i^{\,t}D'_i = \vec{0}, \text{ where } 1 \leq i \leq g \text{ and } \vec{p}_i^{\,t}\vec{d}_i > 0. \qquad (5)$$

The idea is to choose $\vec{p}_i$ as the basis vector of the left null space of matrix $\mathbf{D}'_i$ and invert the sign of the elements of $\vec{p}_i$ if $\vec{p}_i^{\,t}\vec{d}_i < 0$. Hence, by construction, the partitioning matrix $\mathbf{P}$ is feasible.

COROLLARY 1. *If the columns of matrix* $\mathbf{D}'$ *form a normal basis; (i.e., $\vec{d}_i^{\,t}\vec{d}_j = 0$, $i \neq j$), then* $\mathbf{P} = \mathbf{D}'$ *produces a valid partitioning matrix. (See Appendix A.2 for the proof.)*

The following examples illustrate the partitioning scheme and the fact that regularization of a DG into an MMG leads to a uniform way of partitioning the input algorithm.

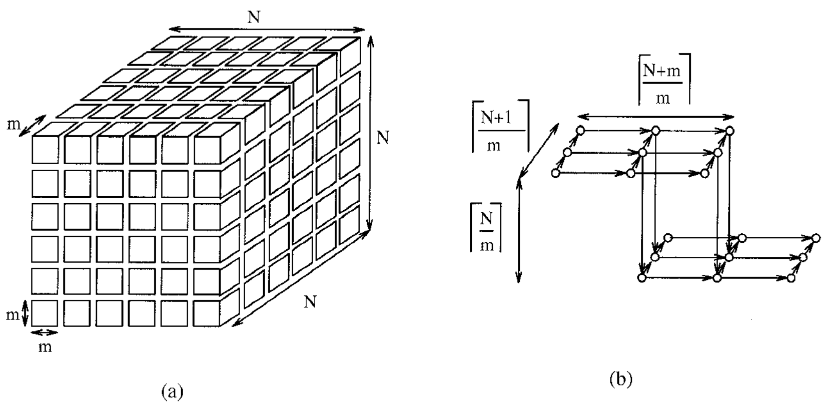

N
m
N
N
m
m
(a)
$\left\lceil \frac{N+m}{m} \right\rceil$
$\left\lceil \frac{N+1}{m} \right\rceil$
$\left\lceil \frac{N}{m} \right\rceil$
(b)

Fig. 7. (a) Partitioned DG for computing matrix products, where partitioning matrix **P** is equal to dependence matrix **D**. (b) Partitioned DG for computing transitive closures, where partitioning matrix **P** is equal to the identity matrix $\mathbf{I}_3$ (the figure is drawn for $N = 4$ and $m = 2$).

EXAMPLE 4. For a 3D matrix product, the dependency matrix is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

According to the procedure above, the partitioning matrix $\mathbf{P} = \mathbf{D}$ is feasible, as $\mathbf{D}^t\mathbf{D} = \mathbf{I}_3 > \bar{0}$. ($\mathbf{I}_3$ is a three-by-three identity matrix.) Fig. 7a shows the partitioning of the DG by the partitioning matrix $\mathbf{P} = \mathbf{D}$. The block-level DG is a cubical 3D mesh of size $\left(\left\lceil \frac{N}{m} \right\rceil\right)^3$.

For the MMG of the transitive-closure problem, the dependence vectors are $(1, 0, 0)^t$, $(0, 1, 0)^t$, $(0, 0, 1)^t$. Assuming that **P** is equal to $\mathbf{I}_3$, Fig. 7b shows the block-level MMG of size $\left\lceil \frac{N+m}{m} \right\rceil \times \left\lceil \frac{N+1}{m} \right\rceil \times \left\lceil \frac{N}{m} \right\rceil$ ($N = 4$, $m = 2$ in this figure). To arrive at this result, note that each plane in Fig. 4b is a square array of $N \times N$ nodes that are padded with one extra row/column of delay nodes for regularization. Hence, each plane has $(N + 1) \times (N + 1)$ nodes, and there are N such planes in the DG. When the DG is partitioned into cubical blocks of size $m \times m \times m$, there will be $\left\lceil \frac{N}{m} \right\rceil$ nodes in the z-direction in the block-level DG and $\left\lceil \frac{N+1}{m} \right\rceil$ nodes in the y-direction. Also, since the DG is staggered from one plane to the next as in Fig. 4b, grouping m such planes together will lead to $\left\lceil \frac{N+m}{m} \right\rceil$ nodes in the x-direction.

The following lemma (stated without proof [10]) shows that the above choice of **P** is good in the sense that it minimizes the amount of data communication between blocks.

LEMMA 2. *The choice of the partitioning matrix* **P** *by Procedure 1 results in the minimum amount of communication between blocks in the partitioned DG.*

The complexity of finding matrix **P** is $\binom{k}{g} O(n.g^2)$ as there are $\binom{k}{g}$ ways of choosing g independent columns of D, and $O(n.g^2)$ is the cost of finding a null-space vector of an $n \times (g - 1)$ matrix.

In general, the size of a block is $b_1 \times b_2 \times \cdots \times b_g$, where b_i, $i = 1, \cdots, g$, and is chosen such that the entire block can be processed by PA in a single pass. The choice is based on the size of PA and the local memory in each PE. In the matrix-product and transitive-closure examples studied in this paper, $g = 3$ and $b_1 = b_2 = b_3 = 3$.

4.2 Executing a Block in PA

The goal in this step is to map a single block onto PA so that it can be executed in a single pass. Here, we use the *Generalized Parameter Method* (GPM) we have developed before [11], [12], [9] to determine the data distribution of inputs of the block chosen for execution; i.e., which data should be input into the boundary PEs at each time step. (Details of GPM are not critical in this paper and are not presented here.) The objective when we apply GPM is to maximize the utilization of PA. If $\#PE$, T_c, and T_{seq} denote, respectively, the number of PEs in PA, completion time of all of the blocks, and serial time to compute the DG, then utilization U is given by

$$U = \frac{T_{seq}}{\# PE \times T_c}. \tag{6}$$

This indicates that, for a given algorithm with fixed T_{seq}, maximizing U is equivalent to minimizing $\#PE \times T_c$. This objective tries to reduce the computation time of each block and increase the overlap between consecutive blocks in order to reduce the load/drain penalties of the blocks.

4.3 Sequencing Blocks through AU

When the recurrence is computed by a CoP, p blocks from MM are initially loaded into AU, and PA begins executing these p blocks. As the execution proceeds, new blocks are fetched from MM into AU, and some of the existing blocks in AU are written back to MM. As AU is of limited size, a block of data will have to be fetched multiple times from MM, and our goal in this step is to decide which blocks will be fetched into AU as execution proceeds such that the amount of memory traffic is minimized.

Previous works on loop transformations or compiler optimizations to improve data locality of loop nests, e.g., [31], [4], target at loops of general dependence structures. These contribute necessary and sufficient conditions of valid loop transformations, but are only heuristic transformations that aim to reduce the traffic in a memory hierarchy. In this study, instead of studying applications with general dependence structures, our sequencing procedure described below is for computations with dependence structures of MMGs. Despite the more limited focus of our work, we show at the end of this section that our sequencing procedure for MMGs is asymptotically optimal in minimizing MM traffic.

The blocks to reside in AU must be chosen to reduce the traffic between MM and AU; equivalently, data reuse should be maximized for the blocks in AU. We can think of AU forming a "storage window" or tile over the block-level DG of the application algorithm. AU stores all the data needed to compute the blocks in the "storage window," and the output of this phase is to describe how the "storage window" will move over the block-level DG in a non-overlapped fashion. (Otherwise, some computations will be redundant.)

The sequencing procedure is described below for a block-level DG in the form of an n-dimensional MMG with identity dependence matrix; i.e., $\mathbf{D} = \mathbf{I}_n$. Let the size of the block-level DG be $V = N_1 \times N_2 \times \cdots N_n$, where N_i, $i = 1, \ldots, n$, is the number of nodes along direction

$$d_i = \{\underbrace{0 \ldots 0}_{i-1}\ 1\ \underbrace{0 \ldots 0}_{n-i}\}^t.$$

The "storage window" is an $(n - 1)$-D tile in the n-D DG.

The following pseudocode describes the movement of the "storage window" of p blocks. Without loss of generality, assume that $N_1 \le N_2 \ldots \le N_n$ (otherwise, the DG can be reindexed).

PROCEDURE 2.

```
for $i_1 = 1$ to $N_1$ step $\sqrt[n-1]{p}$
  ⋱
    for $i_{n-1} = 1$ to $N_{n-1}$ step $\sqrt[n-1]{p}$
      for $i_n = 1$ to $N_n$
        Schedule$(i_1, i_2, \ldots, i_n)$
```

where Schedule($i_1, \ldots, i_n$) schedules all the p blocks in the

$$\overbrace{\sqrt[n-1]{p} \times \ldots \times \sqrt[n-1]{p}}^{n-1}$$

$(n - 1)$-D "storage window" at node $(i_1, \ldots, i_n)$ of the block-level DG to be brought into AU. The "storage window" is on a plane perpendicular to vector d_n in the block-level DG, and moves along direction $\vec{d}_n$.

The reasoning behind Procedure 2 is explained as follows. Consider a general URE in an n-D domain with dependence vectors $\mathbf{D} = [\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_n]$. (If the number of dependencies is larger than n, we consider only the n linearly independent ones.) Note that a set of n linearly independent dependence vectors $\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_n$ can be converted to a set of n unit vectors (corresponding to an n-D mesh) by a linear transformation or an appropriate basis change. Thus, the UREs are, in some sense, equivalent to each other from the point of view of dependencies, and a sequencing scheme developed for n-D MMGs can be extended easily to other UREs. In the case of general UREs, the storage window is an $(n - 1)$-D tile formed by the first $(n - 1)$ dependence vectors, and the tile is moved along the remaining vector $\vec{d}_n$ in the DG.

The shape of a domain of a given URE is also taken into account in the sequencing scheme as follows. For a URE defined over an arbitrary convex domain, the sequencing procedure traverses the domain by a set of parallel 1D "lines." The "width" of a line corresponds to an $(n - 1)$-D storage tile, and its direction denotes the movement of the $(n - 1)$-D tile along the nth dependence vector. A penalty is incurred each time the storage tile shifts from the tail of a line to the head of another parallel line (the term denoted by "XY plane except (0, 0, 0)" in (10) in the following section). The head, tail, and length of these parallel lines are different for different domain shapes. For an n-D mesh, all of the lines are parallel to vector $d_n = (0, \cdots, 0, 1)^t$, and are of the same length N_n. Thus, the regularization of a given DG into an MMG and the decomposition of the block-level DG into parallel lines present a uniform way of handling general UREs.

EXAMPLE 5. Fig. 8a shows the storage-window movement for computing a matrix product with $N = 6m$, $p = 9$, where each tile is a 3×3 square. As the block-level DG is a full 3D mesh, it is perfectly tiled with 216 nodes covered in 24 tiles of size 9 each.

Fig. 8b shows the storage-window movement for computing a transitive closure with $N = 4m$, $p = 4$, where each tile is a 2×2 square. As the block-level DG is not a full 3D mesh, the 100 nodes in the block-level DG are fitted into 36 tiles of four nodes each.

Note that Fig. 8b shows two rows of tiles; hence, there are four rows of blocks in the z-direction (since each tile is a 2×2 block). As each block corresponds to a cubical mesh of $m \times m \times m$ nodes of the original DG, there are $N = 4\ m$ rows in the original DG. Finally, the block-level DG has $\left\lceil \frac{N+m}{m} \right\rceil$ nodes in the x-direction (Fig. 7b), and the block-level DG is staggered from plane to plane. Hence, when tiling it, we are grouping $\sqrt{p}$ nodes in the x and y directions into a tile, and there will be $\left\lceil \frac{N+m}{m} \right\rceil + \sqrt{p} - 1$ tiles in the x-direction.

The size of the storage tile is chosen to minimize the total number of data accesses over the link to MM. Since AU size is constant, the number of data accesses is proportional to the perimeter of the tile.[2] It can be easily seen that the shape

2. In general, the number of data accesses (which is the number of dependence vectors crossing the boundary of a tile) is not proportional to the perimeter and is hard to characterize mathematically. For cubical DGs (and n-D MMGs), since each node at the boundary of a tile has an incoming or outgoing dependence vector (the four corner nodes have both), the number of accesses is dependent on the perimeter. It is easy to construct cases in which all nodes on the periphery do not lead to any access; hence, the number of accesses in general is not proportional to the perimeter.

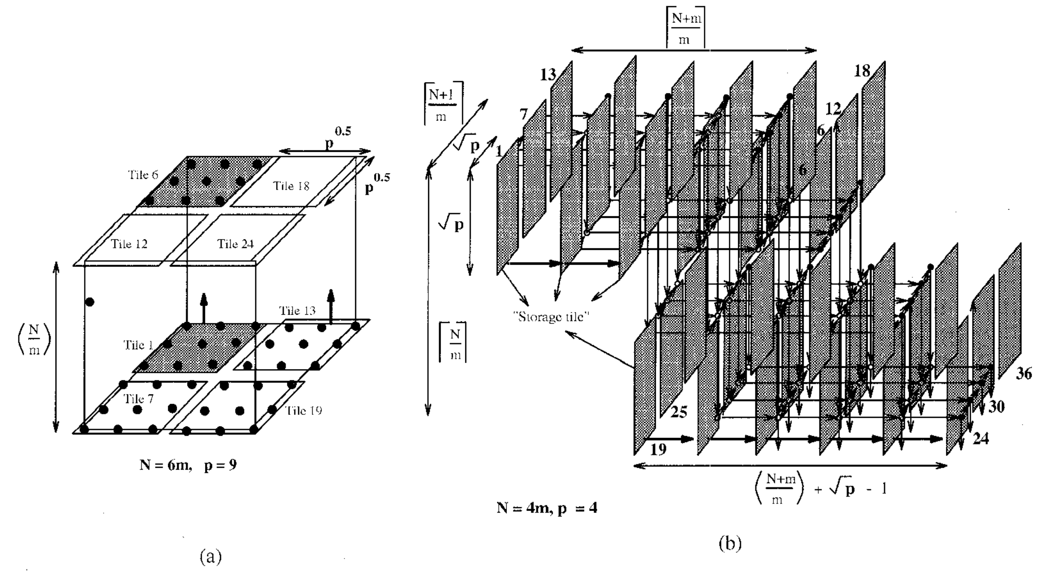


Fig. 8. Storage-window movement in the block-level DG for computing (a) matrix product and (b) transitive closure. The shaded areas show the storage windows or "tiles." For (a), $N = 6m$, $p = 9$, and there are 24 tiles. For (b), $N = 4m$, $p = 4$, and there are 36 tiles.

of the tile should be chosen as an equisided $(n - 1)$-D "parallelepiped" to minimize the total number of data accesses to MM.

For the above sequencing scheme with an equisided tile, the number of data accesses from MM (or the I/O complexity denoted by Q) for a block-level DG in the form of an n-D mesh is

$$Q = \frac{(n-1)V}{m \sqrt[n-1]{p}} + \frac{V}{N_n}, \qquad (7)$$

where V is total number of index points in the n-D DG, and N_n is the size of the largest dimension. Equation (7) can be obtained by considering the inputs along the n faces of an n-D MMG. Since the storage tile is moved along direction $\vec{d}_n$ (Procedure 2), none of the data on the face perpendicular to $\vec{d}_n$ is refetched, and the total number of accesses for inputs on this face is $\frac{V}{N_n}$. For inputs on the other faces perpendicular to $\vec{d}_i$, $i = 1, \ldots, n$, each input data is fetched $\frac{N_i}{m \sqrt[n-1]{p}}$ times from MM. Hence, the total number of accesses for the $(n - 1)$ faces perpendicular to $\vec{d}_i$ is $\frac{V}{N_i} \times \frac{N_i(n-1)}{m \sqrt[n-1]{p}} = \frac{V(n-1)}{m \sqrt[n-1]{p}}$.

The following lemma establishes that the above sequencing scheme for n-D MMGs is asymptotically optimal with respect to the number of accesses to MM.

LEMMA 3. *For n-dimensional MMGs, $Q = \Omega\left(\frac{V}{\sqrt[n-1]{S}}\right)$, where S is the size of the limited memory and Q is the I/O complexity. (See* [13] *for proof.)*

In this case, $S = p$, and the number of accesses from MM ((7)), due to Procedure 2 which has the same asymptotic complexity as the lower bound given in Lemma 3. (Note that the factor $(n - 1)$ is constant for a specific MMG and does not affect the complexity.) Hence, our sequencing scheme is optimal in terms of the number of accesses from MM.

5 Applications: Matrix Product and Transitive Closure

In this section, we present our results in mapping algorithms described by MMGs on a CoP. Our results are based on the matrix-product and transitive-closure applications. Figs. 4, 7, and 8 show, respectively, the original MMG, the block-level DG after partitioning, and the movement of the tiles for these two problems.

For the matrix-product problem, let $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ has a 3D MMG of size $N \times N \times N$, where $\mathbf{A}$ and $\mathbf{B}$ are two $N \times N$ matrices. As shown in Section 4.1, this DG is partitioned into cubical blocks of size $m \times m \times m$. The AU holds a $\sqrt{p} \times \sqrt{p}$ square tile of blocks of DG (formed along dependencies $(1, 0, 0)^t$ and $(0, 1, 0)^t$), and the tiles move along dependence vector $(0, 0, 1)^t$.

For the transitive-closure problem, darkened nodes representing delay nodes are first added to regularize the original DG to a cubical structure (Fig. 4b). The 3D MMG is then partitioned into cubical blocks of size $m \times m \times m$, each of which can be executed in one pass on PA. The size of the block-level DG after partitioning is $\left\lceil \frac{N+m}{m} \right\rceil \times \left\lceil \frac{N+1}{m} \right\rceil \times \left\lceil \frac{N}{m} \right\rceil$. Note that the block-level DG is identical to the original DG except for its dimensions. The "storage window" has size $\sqrt{p} \times \sqrt{p}$ blocks and moves along direction $(0, 1, 0)^t$ as shown in Fig. 8b in order to minimize the total number of data accesses.

These examples illustrate the benefit of deriving MMGs from nested-loop algorithms, as we have a uniform way of partitioning an MMG, sequencing blocks of the MMG, and designing a PE array to execute a block of the MMG.

5.1 Evaluation Metrics

To develop a cost-effective design, we need to evaluate the performance of the target algorithms on a given amount of silicon chip area. In this subsection, we present an abstract model to estimate the area consumed and the completion time of execution.

5.1.1 Area Model

The total area occupied by the CoP architecture is the sum of the areas of the PEs (including input/output logic), AU, controller, and input and output networks in AU:

$$Area = Area_{PA} + Area_{AU} + Area_{pins} + Area_{controller} + Area_{network}. \quad (8)$$

The last three terms can be assumed to be constants. This assumption is especially true for PAs with linear configurations because the number of ports in such PAs and AUs are constant. The analysis in the rest of this section will still be valid.) If the values of $Area_{pins}$, $Area_{controller}$, and $Area_{network}$ are small, as compared to the values of $Area_{PA}$ and $Area_{AU}$, then the area index containing the dominant terms is given by

$$AreaIndex = \underbrace{\#\,PE\left(A_{pe} + b\right)}_{Area_{PA}} + \underbrace{3\sqrt{p}m^2 + pm^2}_{Area_{AU}}, \quad (9)$$

where A_{pe} is the area of a single PE in memory words (which captures the implementation cost of a PE), and b reflects the local memory per PE. Since in our examples AU holds $\sqrt{p} \times \sqrt{p}$ blocks of the DG forming a square in the 3D MMG (Fig. 8), storage is needed in AU for

1) pm^2 words of C,
2) $\sqrt{p}m^2$ words of A,
3) $\sqrt{p}m^2$ words of B, and
4) additional $\sqrt{p}m^2$ words for the next set of $\sqrt{p} \times \sqrt{p}$ blocks to be brought in from MM to AU for future processing.

In the following sections, we use the simplified area model as a cost index to analyze the cost-performance relation of the CoP architecture. It is, however, important to point out that in some cases $Area_{controller}$ may be large and need to be considered in computing the area. Similarly, $Area_{network}$ and $Area_{pins}$ may depend on word size, number of buffers, and number of paths to PA. The net effect of including these additional areas in our area model is to shift all the cost-performance curves in Figs. 10 and 11 to the right. The analysis is similar and will not be shown.

5.1.2 Model of Completion Time

The total completion time T_{compl} in PE-cycles is given by

$$T_{compl} = max\left(\frac{2\sqrt{p}m^2}{B_{MM}}, pt_{block}\right)\left(\frac{K(N,m)}{p} - 1\right) + \underbrace{\frac{2pm^2}{B_{MM}}\left(\frac{K_{XY}(N,m)}{p} - 1\right)}_{XY\ plane\ except\ block(0,\,0,\,0)} + \underbrace{\frac{pm^2 + 2\sqrt{p}m^2}{B_{MM}}}_{block(0,\,0,\,0)} + \underbrace{max\left(\frac{pm^2}{B_{MM}}, (p-1)t_{block}\right)}_{block(N,N,N)} \quad (10)$$

where B_{MM} is the MM-AU bandwidth in words per PA clock tick, t_{block} is the nonoverlapped time (total completion time—portions of the load/drain times overlapped by the following/preceding blocks) taken to execute one block of DG by PA,[3] $K(N, m)$ is the number of blocks of the DG or the number of nodes in the block-level DG, and $K_{XY}(N, m)$ is the number of nodes in the projection of the block-level DG to the XY plane parallel to the storage tile of AU; i.e., $K_{XY}(N, m)$ is the number of times the storage tile has to change direction. The term "XY plane except block(0, 0, 0)" in (10) means storage tiles that can be fetched without shifting from the tail of a fetch direction to the head of another parallel fetch direction.

The first term in (10) is the dominant term and is the product of the time taken for each window $(\sqrt{p} \times \sqrt{p})$ in AU and the number of windows over DG. For each window of size $\sqrt{p} \times \sqrt{p}$, $\frac{2\sqrt{p}m^2}{B_{MM}}$ is the time it takes to fetch the elements needed for the next window, and pt_{block} is the time to process p blocks in the current window. The second term in (10) models the additional time required whenever the window changes direction, which involves writing and reading pm^2 elements of the output matrix. The third term is the initial latency to load the data corresponding to $\sqrt{p} \times \sqrt{p}$ blocks in the storage window. For a $\sqrt{p} \times \sqrt{p}$ storage window, there are pm^2 elements of the result matrix (corresponding to the area of the tile) and $2\sqrt{p}m^2$ elements of the input matrix (corresponding to half of the perimeter). The final term is the additional time over the time pt_{block} for the results of the final storage tile to be written back from AU to MM. Note that there are pm^2 elements of the output matrix to be written back to MM, and t_{block} is the earliest time after the start of the last storage window when the first m^2 elements are available to be written back.

EXAMPLE 6. For the matrix-product problem, as shown in Fig. 8a,

$$K(N,m) = \left(\left\lceil\frac{N}{m}\right\rceil\right)^3$$

$$K_{XY}(N,m) = \left(\left\lceil\frac{N}{m}\right\rceil\right)^2$$

For the transitive-closure problem with 3D MMG, as shown in Fig. 8b,

$$K(N,m) = \left\lceil\frac{N+m}{m}\right\rceil \times \left\lceil\frac{N+1}{m}\right\rceil \times \left\lceil\frac{N}{m}\right\rceil$$

$$K_{XY}(N,m) = \left\lceil\frac{N+1}{m}\right\rceil \times \left\lceil\frac{N}{m}\right\rceil.$$

5.1.3 AU Size

In order to have efficient processing, AU must be large enough to mask MM latency fully; i.e., completely overlap

3. For any block of DG, the processing time includes the load, drain, and computation times of the block on PA. In GPM [11], [9], block scheduling is done to overlap consecutive blocks entering PA, thereby reducing the effective load, drain (and maybe the computation time) of a block. The nonoverlapped time for a block refers to those portions of the load, drain, and computation times that are not masked by successive blocks.

the loading of the inputs of the next storage window with the processing of the current one. Therefore,

$$\frac{2\sqrt{p}m^2}{B_{MM}} \leq pt_{block} \Rightarrow p \geq \left(\frac{2m^2}{B_{MM}t_{block}}\right)^2, \tag{11}$$

and AU size is computed using (9) for this value of p.

EXAMPLE 7. Let $N = 512$, $m = 8$, and $B_{MM} = 1/5$ (five cycles per word). The number of processors, $\#PE = m^2 = 64$.

- For a square array, $t_{block} = m = 8$. By (11), $p \geq 6400$, and the minimum AU size is about 425*K* words or 13.3*M* bits, assuming four bytes/word.
- For a linear PA with $t_{block} = m^2 = 64$. By (11), $p \geq 100$, and the minimum AU size is about 8.3*K* words or 260*K* bits.

Although 13*M* bits of fast memory in AU is at the limit of current technology, 260*K* bits is very feasible. For a linear PA, as $t_{block} = O(m^2)$, p is independent of m. This is true because, for a linear PA used for block processing, its I/O bandwidth is constant, independent of the number of PEs and block size, and AU size in blocks (p) depends only on the bandwidth between MM and AU.

5.1.4 Performance Metrics

The performance index of a CoP is defined as its speedup over a reference design:

$$PerformanceIndex = \frac{T_{compl}^{ref}}{T_{compl}^{cop}} \tag{12}$$

where T_{compl}^{ref} and T_{compl}^{cop} are the total completion times in PE-cycles of the reference design and the CoP, respectively. A CoP with one PE and an appropriate amount of AU memory to mask MM latency is chosen as the reference design. Thus, both the reference and current designs have the same bandwidth limitation. The *AreaIndex* of the one-PE reference design in memory words can be obtained from (9) and (11) when $m = 1$, $t_{block} = 1$, and $\#PE = 1$. It is given by

$$AreaIndex_{ref} = A_{pe} + \frac{4}{B_{MM}^2} + \frac{6}{B_{MM}}. \tag{13}$$

A useful trade-off in designing a CoP is as follows. Suppose we increase area by a certain factor, how much will speedup be increased (or how much will completion time be reduced) if we clock the reference machine and the CoP at the same rate? The speedup over the reference design can be interpreted as the reduction in clock rate in order to obtain the same throughput (or completion time). In other words, we have

$$T_{compl} \times \frac{1}{Clock_{PA}} = T_{ref} \times \frac{1}{Clock_{ref}} \Rightarrow \frac{Clock_{ref}}{Clock_{PA}} = \frac{T_{ref}}{T_{compl}} \tag{14}$$

where T_{compl}^{ref} and $Clock_{ref}$ are the completion time and clock rate of the reference design, respectively. Sections 5.4 and 5.5 present cost-performance trade-offs of CoPs where cost is measured as *AreaIndex* defined in (9), and performance is measured as speedup over the one-PE reference design ((12)).

A reduced clock rate is desirable for several reasons. First, and most important, the yield would be significantly higher if the chip were designed for a lower clock rate. Also, power dissipation is lower at lower clock rates, leading to lower packaging and integration costs.

5.2 Relationship between Performance and Area Indices

For a 3D MMG, if the size of PA is increased by a factor θ, then block size m will have to be increased by θ for single-pass execution of each block by PA. For a linear PA, $t_{block} = O(m^2)$, and the number of blocks in AU (p) is independent of block size m ((11)). Therefore, the area of AU for a linear PA increases by θ^2 when m is increased by θ ((9)). The total area (*AreaIndex*) of the CoP, which is dominated by the area of AU, increases by a factor θ^2 when m is increased by θ. Hence, *AreaIndex* grows as the square of the number of PEs in a linear PA. The following argument shows that AU size has to grow at least as the square of the number of PEs in order to mask MM latency when processing a cubical block of DG on a linear PA.

Consider a cubical $m \times m \times m$ block of DG to be processed in a linear PA. For a linear PA to have constant I/O bandwidth, the time to process a block is $\Omega(m^2)$, as there are $O(m^2)$ input and output elements to be loaded into PA. Therefore, AU size has to be $\Omega(m^2)$ as all the $O(m^2)$ elements needed to process a block have to be held in AU to mask the MM latency. The number of PEs in PA is $O(m)$, as there are m^3 operations to be completed in $O(m^2)$ time. Therefore, to mask MM latency, AU size (given by $\Omega(m^2)$) grows at least as the square of the number of PEs (given by $O(m)$).

The completion time T_{compl} can, at best, decrease by θ when the number of PEs is increased by θ (superlinear speedups are not possible for deterministic processing). The area index grows at least as θ^2 when the number of PEs is increased by θ. Thus, clock-rate reduction can grow at best as the square root of *AreaIndex* when the effect of MM latency is masked completely. However, beyond a certain number of PEs (or *AreaIndex*) for a fixed problem size, the completion time is bounded by the fixed MM bandwidth and is equal to the time to read and write the elements of input and output matrices. Hence, speedup will flatten out beyond a certain *AreaIndex* (A_{crit}) when MM becomes a bottleneck. That is,

$$\text{Speedup} = \frac{T_{compl}^{ref}}{T_{compl}^{cop}} = \begin{cases} O\left(\sqrt{AreaIndex}\right) & \text{for } AreaIndex \leq A_{crit} \\ \dfrac{Volume(N)}{B_{MM}} & \text{for } AreaIndex > A_{crit} \end{cases} \tag{15}$$

where *Volume*(N) is the total amount of data in the 3D MMG to be accessed (including both reads and writes) to process the given n-dimensional uniform dependence algorithm.

In short, if the completion time of the reference PA is fixed, then we have the following inverse square-root relationship between the completion time of PA and its *AreaIndex* for fixed MM bandwidth and specified clock rate in processing a 3D MMG.

$$T_{compl}^{PA} \propto \frac{1}{\sqrt{AreaIndex_{PA}}}, \text{ for } AreaIndex_{PA} \leq A_{crit} \tag{16}$$

5.3 Area Allocation between AU and PA

Fig. 9 shows the variation of the fraction of total area occupied by PA as we increase the total area of the CoP. The AU area in (9) is computed using p from (11), which is chosen to overlap memory fetches completely with computations. Note that the plot is dependent on p and *AreaIndex* but not on the application problem and its problem size. The x-axis is the area of the chip in memory words. Thus, *AreaIndex* of five megawords corresponds to an area equivalent to 160 megabits of storage, assuming each word is 32 bits. The cost of a PE in memory words is denoted as A_{pe}. Hence, for five megawords of total area and A_{pe} = 100, only four percent of the total area is occupied by PEs for a linear PA configuration. This shows that most of the chip area is taken up by AU if we design the chip with the optimal balance where memory latency is fully masked. Moreover, for the same total area, a linear PA has more of its area devoted to PEs than a PA in the form of a square mesh. The effect of increased area of a PE is to lift the entire plot upwards for both linear and square PAs. Although this figure is for B_{MM} = 1/5 (five cycles to access a word from memory), the same effect is observed for other bandwidths.

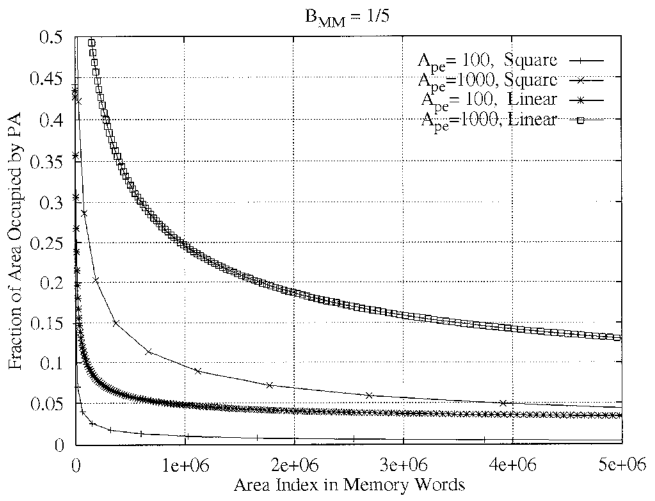


Fig. 9. Area allocation between AU and PA of a CoP for masking MM latency fully. The area of the one-PE reference design is equal to 230 words for A_{pe} = 100, and 1,130 words for A_{pe} = 1,000 ((13)).

5.4 Cost-Performance Trade-Offs: Matrix Product

Fig. 10a shows the cost-performance trade-offs of CoPs for computing matrix products described by 3D MMGs. Performance is measured in speedup (or clock rate), and cost is measured as *AreaIndex*. The system is designed at the balance point to mask MM latency fully ((11)). Trade-offs are shown for problem sizes of N = 512 and N = 1,024 when the latency to access a word from MM is five cycles. The area cost of a PE in memory words is denoted as A_{pe}. The amount of local memory in each PE is controlled by parameter b. In our approach, for a given block size, we obtain a virtual array (linear or square), and cluster b virtual PEs to obtain increased local memory per PE and reduced number of PEs. Hence, the larger is the value of b, the lower are the number of PEs and clock-rate reduction. For b = 1, each physical (and virtual) PE has three words of storage, one for **A**, **B**, and **C**, respectively.

Fig. 10a shows that for about 1.1×10^6 words of silicon area and A_{pe} = 100, we can get a speedup of about 51 over the one-PE reference design for a square PA and 48 for a linear PA in computing a 1,024-by-1,024 matrix product. In other words, if we clock a CoP with a linear PA at 400 KHz, we will obtain performance equivalent to that of a one-PE reference design running at 19.2 MHz. If we can clock this CoP at 1 MHz, then we will obtain 2.5 times speedup as compared to the one-PE reference design. The final speedup can be chosen from a variety of alternatives depending on the objective of the design.

Fig. 10a further shows the square-root relationship between speedup and *AreaIndex* (Section 5.2). For instance, to compute a 1,024-by-1,024 matrix product using a linear PA with A_{pe} = 5,000, the speedup only doubles from 20 to 40 when *AreaIndex* is increased from 0.45 megawords to 1.5 megawords. The degradation in performance in using a linear PA rather than a square array when AreaIndex is about 1.1 megawords is six percent for A_{pe}= 100 and 26 percent for A_{pe} = 5,000.

Fig. 10a also shows that performance of a CoP saturates because of the MM bottleneck with limited bandwidth. For smaller problems, speedup will flatten out earlier. After the performance saturates, the completion time of the CoP is equal to the time to access input matrices **A** and **B** from MM and to write output matrix **C** to MM. Hence, the maximum speedup (or clock-rate reduction) is given by $\frac{T^{ref}_{comp}}{T^{cop}_{comp}} = \frac{Clock_{ref}}{Clock_{cop}} = \frac{N^3}{\frac{4N^2}{B_{MM}}} = 51.2$ for N = 1,024. This maximum value of 51.2 corresponds to *AreaIndex* $\geq$ 1.44 megawords (A_{crit} = 1.44 megawords) when using a linear PA and A_{pe} = 100.

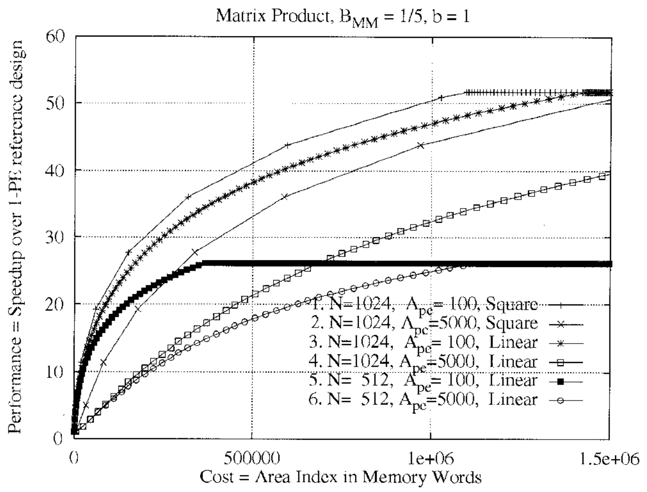
The saturation points improve as products of larger matrices are computed. Our earlier work [11], [9] shows that speedup does not saturate in the range evaluated in Fig. 10 when N = 10,240.

The results in Fig. 10a show that a linear PA is an attractive choice for computing matrix products and other UREs. Its advantages are its constant I/O bandwidth and modularly expandable layout. It achieves good performance because our mapping algorithm can exploit effectively locality in the three-dimensional loops.

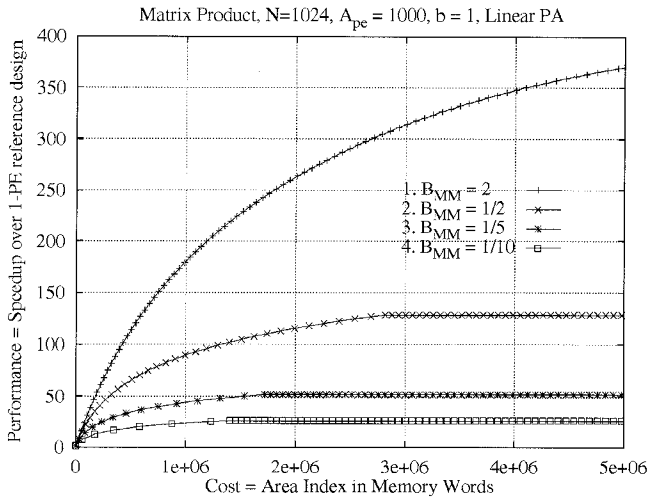
Fig. 10b depicts the performance for linear and square PAs with b = 8. Here, eight virtual PEs are coalesced into a physical PE leading to an eight-fold increase in local memory of a PE. While the corresponding curves for square PAs remain unchanged from Fig. 10a, the curves for linear PAs are lower than the corresponding ones in Fig. 10a. Again, speedup grows as the square root of *AreaIndex* when MM latency is masked.

Fig. 10c shows the effect of increasing B_{MM} between MM and AU. The figure shows that MM bandwidth is a key factor that influences the performance of CoPs. For instance, when *AreaIndex* is five megawords, performance doubles from 51.2 to 128 when B_{MM} is increased from 1/5 (five cycles/word) to 1/2 (two cycles/word).

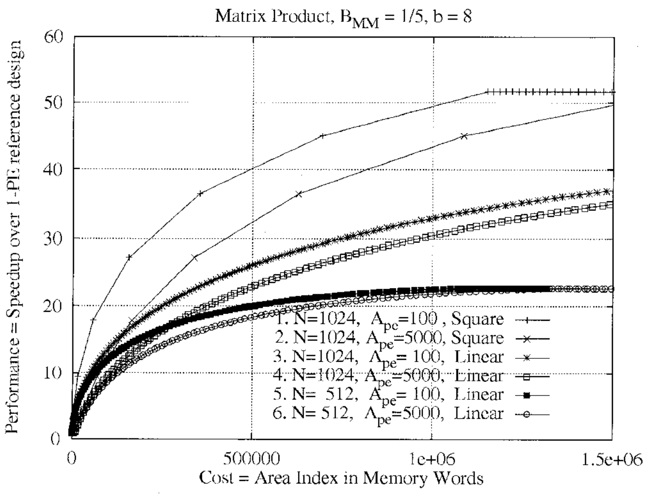
Finally, Fig. 10d shows the sensitivity of speedup to variations in AU size. Assume that AU size is scaled by a



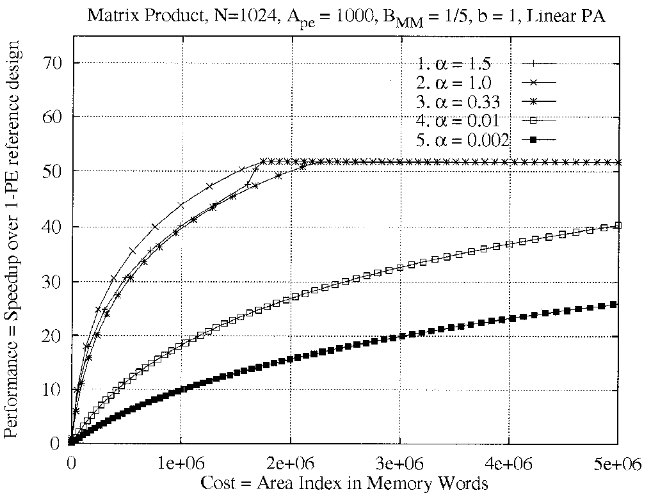
(a) Cost-performance trade-offs. The values of maximum speedup are: $SU^1_{max} = SU^2_{max} = SU^3_{max} = SU^4_{max} = 51.2$, $SU^5_{max} = SU^6_{max} = 25.6$. The values of critical *AreaIndex* in megawords beyond which speedup saturates are: $A^1_{crit} = 1.1$, $A^2_{crit} = 1.70$, $A^3_{crit} = 1.44$, $A^4_{crit} = 2.97$, $A^5_{crit} = 0.368$, $A^6_{crit} = 1.14$.



(c) Effect of bandwidth (B_{MM}) on cost-performance trade-offs. The values of maximum speedup for plots 2, 3, 4 are: $SU^2_{max} = 128$, $SU^3_{max} = 51.2$, $SU^4_{max} = 25.6$. The values of $A^2_{crit} = 2.80$, $A^3_{crit} = 1.73$, $A^4_{crit} = 1.40$. Plot 1 does not saturate because the MM bandwidth is higher than the data rate required by PA.



(b) Effect of *b* (local memory per PE) on cost-performance trade-offs. Local memory in a PE is used to simulate a set of virtual PEs and to reduce the PE-count. The values of maximum speedup are: $SU^1_{max} = SU^2_{max} = 51.2$, $SU^3_{max} = SU^4_{max} = 44.9$, $SU^5_{max} = SU^6_{max} = 22.7$. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = 1.16$, $A^2_{crit} = 1.76$, $A^3_{crit} = 4.24$, $A^4_{crit} = 4.75$, $A^5_{crit} = 1.07$, $A^6_{crit} = 1.33$.

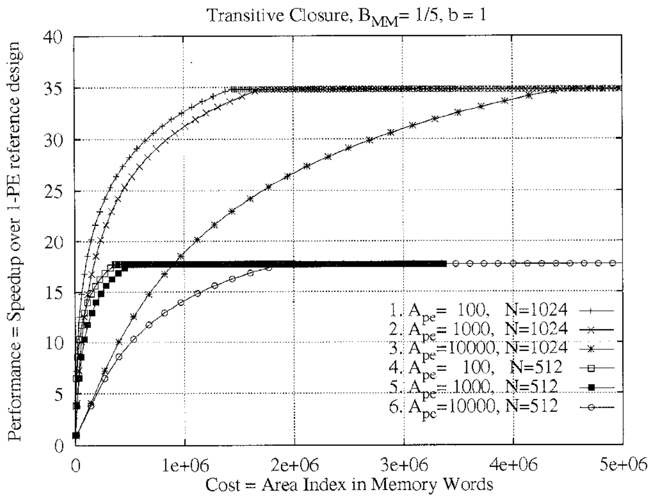


(d) Sensitivity of cost-performance trade-offs for the matrix-product problem when the optimal AU size is multiplied by α. If $\alpha < 1$, all the PEs are idle for some time due to insufficient data accessed from MM. If $\alpha > 1$, chip area is wasted by the extra memory in AU. The maximum speedup for all curves is 51.2. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = A^2_{crit} = 1.72$, $A^3_{crit} = 2.24$, $A^4_{crit} = 10.4$, $A^5_{crit} = 30.6$.

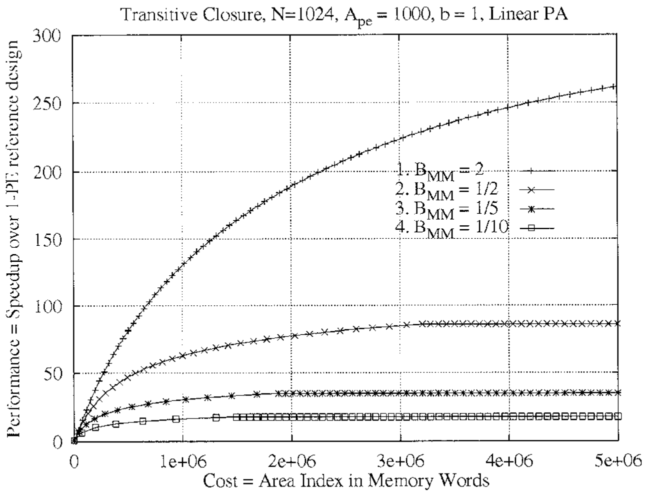
Fig. 10. Evaluation of CoPs for computing matrix products.

factor of α from the optimal value given in (11). If $\alpha < 1$, all PEs will be idle between the time the current set of p blocks are completed to the start of the next set of p blocks. Therefore, speedup (or clock-rate reduction) ((15)) will decrease as α is decreased.

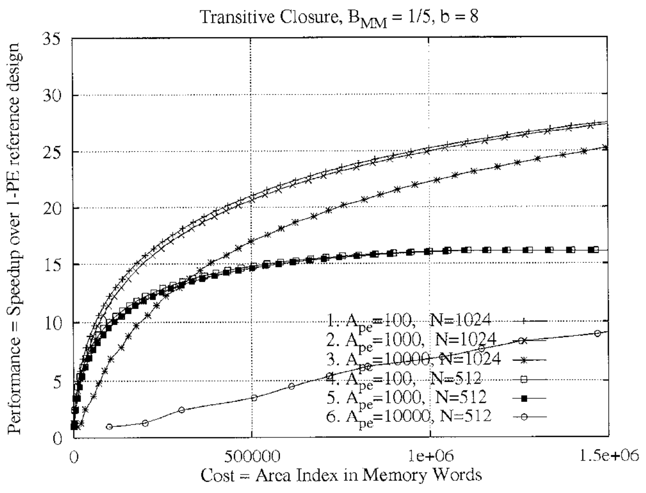
If $\alpha > 1$, it means that less area will be allocated to PA for a fixed chip area. Since PA is already working at the maximum efficiency, the additional area allocated to AU cannot help boost the performance of the CoP, resulting in a performance curve that is below the performance curve when $\alpha = 1$. When $\alpha > 1$ and AU size is larger than the total number of words in the input matrices, the extra area allocated to AU beyond that needed by the input matrices will be unused. In this case, we set AU size to the maximum size needed by the input matrices. This explains why the critical *AreaIndex* A_{crit} in Fig. 10d is the same for all $\alpha \geq 1$, and why



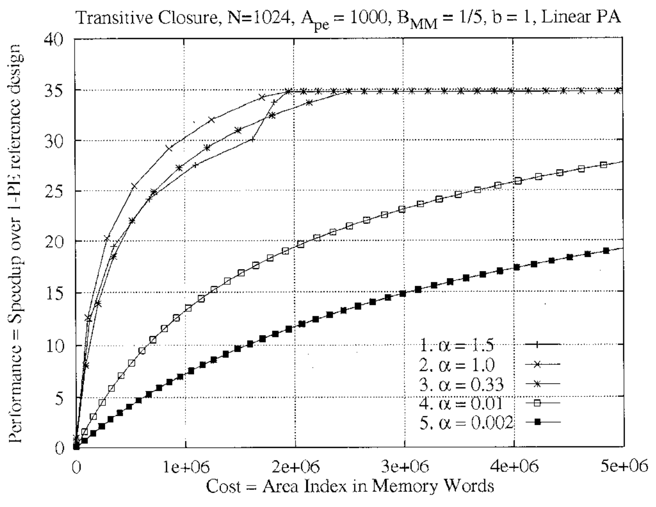
(a) Cost-performance trade-offs. The values of maximum speedup are: $SU^1_{max} = SU^2_{max} = SU^3_{max} = 34.8$, $SU^4_{max} = SU^5_{max} = SU^6_{max} = 17.7$. The values of critical *AreaIndex* in megawords beyond which speedup saturates are: $A^1_{crit} = 1.44$, $A^2_{crit} = 1.73$, $A^3_{crit} = 4.6$, $A^4_{crit} = 0.37$, $A^5_{crit} = 0.46$, $A^6_{crit} = 1.9$.



(c) Effect of bandwidth (B_{MM}) on cost-performance trade-offs. The values of maximum speedup for plots 2, 3, 4 are: $SU^2_{max} = 86$, $SU^3_{max} = 34.8$, $SU^4_{max} = 17.7$. The values of critical *AreaIndex* in megawords are: $A^2_{crit} = 3.19$, $A^3_{crit} = 1.89$, $A^4_{crit} = 1.48$. Plot 1 does not saturate as MM bandwidth is higher than the data rate required by PA.



(b) Effect of b (local memory per PE) on cost-performance trade-offs. The values of maximum speedup are: $SU^1_{max} = SU^2_{max} = SU^3_{max} = 31.6$, $SU^4_{max} = SU^5_{max} = SU^6_{max} = 16.1$. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = 4.30$, $A^2_{crit} = 4.42$, $A^3_{crit} = 5.36$, $A^4_{crit} = 1.1$, $A^5_{crit} = 1.1$, $A^6_{crit} = 6.15$.



(d) Sensitivity of cost-performance trade-offs for the transitive closure problem when the optimal AU size is multiplied by α. The maximum speedup for all curves is 34.8. The values of critical *AreaIndex* in megawords are: $A^1_{crit} = A^2_{crit} = 1.88$, $A^3_{crit} = 2.52$, $A^4_{crit} = 12.0$, $A^5_{crit} = 34.1$.

Fig. 11. Evaluation of CoPs for computing transitive closures.

the performance curve for $\alpha > 1$ merges with the optimal performance curve ($\alpha = 1$) at A_{crit}.

Fig. 10d further shows that for A_{pe} = 1,000, *AreaIndex* = 5*M* words, and a 500-fold decrease in AU size ($\alpha = 0.002$), speedup drops from 51.2 to 26. This show that linear-PA designs show good resilience to changes in AU size from their desired optimal configurations. The low sensitivity to AU size can be used to obtain significant area savings without large sacrifices in performance. For instance, when α = 0.33 and *AreaIndex* ≈ 1.7*M* words, AU is one-third smaller than the case when α = 1.0, resulting in 58 percent area savings (from Figs. 9 and 10d) for only eight percent decrease in performance (where speedup is reduced from 51 to 47). This resilience allows designers to fine tune their designs in order to obtain different area-performance trade-offs.

5.5 Cost-Performance Trade-Offs: Transitive Closure

Fig. 11 shows the cost-performance trade-offs in applying CoPs to solve transitive-closure problems described by 3D MMGs. The x-axis is the cost or *AreaIndex* in memory words, and the y-axis is the speedup over the one-PE reference design.

Due to the more irregular nature of the transitive-closure algorithm, the performance of CoPs in computing transitive closures is lower than that in matrix products. For instance, Fig. 11a shows a speedup of only 34.8 over the one-PE reference design (as compared to 51.2 for the matrix-product problem) using a linear PA with five megawords of silicon area, $A_{pe} = 1{,}000$, and $N = 1{,}024$. For the same configuration, Fig. 11b shows that increased local memory in each PE ($b = 8$) results in scaling the performance down to 31.6. Likewise, Fig. 11c shows improved performance when B_{MM} is increased. Finally, Fig. 11d establishes that cost-performance trade-offs are not very sensitive to AU size required for masking MM latency fully. For instance, when $\alpha = 0.33$ and *AreaIndex* = 1.86, AU is one-third smaller than when $\alpha = 1.0$, resulting in a 58 percent area savings for a six percent decrease in performance (where speedup is reduced from 34.8 to 32.7).

As in the case of computing matrix products, the saturation points improve when larger matrices are involved. Our earlier work [11], [9] shows that speedup does not saturate in the range evaluated in Fig. 11 when $N = 10{,}240$.

6 Conclusions

This paper describes the design of a coprocessor (CoP) for executing loop computations described by uniform dependence algorithms. Our results show that high performance is achieved by a modularly expandable linear array of PEs, coupled with local buffer memory that interfaces a Main Memory (MM) with fixed bandwidth. The high performance is achieved by efficient methods for mapping recurrences on CoPs. There are three major contributions in this paper.

- For the class of multimesh dependence graphs, we have developed a partitioning scheme that we have proved to be asymptotically optimal. Other existing schemes have been developed for more general dependence graphs and may be suboptimal with respect to multimesh graphs. It is important to point out that optimality is only asymptotic; that is, for a specific problem instance and Access-Unit (AU) size, it is possible that our partitioning scheme may induce more memory traffic than an existing scheme.
- In computing matrix products and transitive closures, we have found the optimal division of chip area between Processor Array (PA) and AU for given chip size, MM bandwidth, and tiling method. We have also shown area-speedup trade-offs when optimal partitioning of chip area is not done. Such trade-offs are important when combining multiple CoP chips together to form a larger CoP system.
- For dependence graphs represented as three-dimensional multimesh graphs (MMGs) and processor arrays in a linear configuration, our study finds an important relation that performance is related to the inverse square root of cost. Here, cost is measured as silicon area, and performance, as completion time under fixed main-memory bandwidth and constant clock speed ((16)). Another interpretation of performance is in the reduction of clock speed, where increases in *AreaIndex* in (15) can be considered as reduction in clock rate for a given completion time.

  We have demonstrated the inverse square-root relation using the matrix-product and transitive-closure applications. In addition, we have evaluated CoPs of various configurations and have studied the effects of different parameters on the inverse square-root relationship.

Although our results have been derived for three-dimensional multimesh dependence graphs, they can be extended to multimesh graphs of higher dimensions. Given a linear processor array, performance will be related to the inverse rth root of *AreaIndex* where $r + 1$ is the dimension of the multimesh graphs.

There are two areas that we will study in the future.

1) The effectiveness of uniformizing non-uniform recurrences into multimesh graphs. We will need to evaluate alternative multimesh dependence-graph representations of a given dependence graph.
2) The design of a hardware controller and the associated software compiler to support the partitioning and mapping processes.

Appendix

Proof of Lemma 1

A given partitioning matrix **P** is valid if the compressed or block-level DG is acyclic. Hence, given a partitioning vector $\vec{p}_i$, all of the dependence vectors $\vec{d}_j$ should cross the hyperplane corresponding to $\vec{p}_i$ in the same direction; i.e., $\forall j\ \vec{p}_i^t\vec{d}_j \geq 0$ or $\vec{p}_i^t\vec{d}_j < 0$. Fig. 12 shows the case in which $\vec{p}_i$ is not valid (Block 1 depends on Block 2 and vice versa).

Proof of Corollary 1

Consider column $\vec{p}_i$ of partitioning matrix $\mathbf{P} = \mathbf{D}'$, where $\mathbf{D}'$ is an orthonormal matrix. Hence, $\vec{p}_i^t\vec{d}_i = \vec{d}_i^t\vec{d}_i = \left|\vec{d}_i\right|^2 > 0$, where $\left|\vec{d}_i\right|$ is the magnitude of $\vec{d}_i$. By Lemma 1, **P** is valid. Also, $\vec{p}_i^t\vec{d}_j = 0,\ j \neq i$; hence, it satisfies (5).

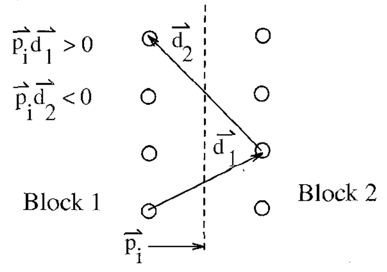


Fig. 12. A pair of cyclically dependent blocks. Block 2 depends on block 1 through dependence $\vec{d}_1$, while block 1 depends on block 2 through $\vec{d}_2$. Hence, partitioning vector $\vec{p}_i$ is invalid.

ACKNOWLEDGMENTS

Research supported by Joint Services Electronics Program contract N00014-90-J-1270, U.S. National Science Foundation grants MIP 92-18715 and MIP 96-32316, and an IBM graduate fellowship grant. A preliminary version containing about 25 percent of this paper appeared in the Proceedings of the IEEE Symposium on Parallel and Distributed Processing, pp. 806-813, December 1993.

REFERENCES

[1] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb, "The Warp Computer: Architecture, Implementation and Performance," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,523-1,538, Dec. 1987.

[2] J. Bu and E.F. Deprettere, "Processor Clustering for the Design of Optimal Fixed-Size Systolic Arrays," *Proc. Application Specific Array Processors*, pp. 402-413, IEEE CS Press, Sept. 1991.

[3] W.P. Burleston, "Partitioning Problem on VLSI Arrays: I/O and Local Memory Complexity," *Proc. of ICASSP*, pp. 1,217-1,220, Toronto, Canada, May 1991.

[4] S. Carr and K. Kennedy, "Compiler Blockability of Numerical Algorithms," *Proc. Int'l Conf. Supercomputing*, pp. 114-124, 1992.

[5] V.V. Dongen, "Mapping Uniform Recurrence onto Small Size Arrays," *Proc. PARLE*, pp. 191-208, 1991.

[6] B.L. Drake, F.T. Luk, J.M. Speiser, and J.J. Symanski, "SLAPP: A Systolic Linear Algebra Computer," *Computer*, vol. 20, no. 7, p. 45, July 1987.

[7] 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.

[8] D.E. Fouler and R. Schreiber, "The Saxpy Matrix-1: A General Purpose Systolic Computer," *Computer*, vol. 20, no. 7, p. 35, July 1987.

[9] K. Ganapathy and B.W. Wah, "Optimal Synthesis of Algorithm-Specific Lower-Dimensional Processor Arrays," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 3, pp. 274-287, Mar. 1996.

[10] K. Ganapathy, "Mapping Regular Recursive Algorithms to Fine-Grained Processor Arrays," PhD thesis, Univ. of Illinois, Urbana-Champaign, May 1994.

[11] K. Ganapathy and B.W. Wah, "Optimizing General Design Objectives in Processor Array Design (extended paper)," *Proc. IEEE Int'l Parallel Processing Symp.*, pp. 295-302, Apr. 1994.

[12] 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.

[13] J.-W. Hong and H.T. Kung, "The I/O Complexity: The Red Blue Pebble Game," *Proc. 13th Ann. ACM Symp. Theory of Computing*, pp. 326-333, May 1981.

[14] F. Irigoin and R. Triolet, "Supernode Partitioning," *Proc. 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, pp. 319-329, Jan. 1988.

[15] K. Jainandunsing, "Optimal Partitioning Schemes for Wavefront/Systolic Array Processors," technical report, Delft Univ. of Technology, Delft, The Netherlands, Apr. 1986.

[16] R.M. Karp, R.E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrences," *J. ACM*, vol. 14, pp. 563-590, July 1967.

[17] P. Kuchibhotla and B.D. Rao, "Efficient Scheduling Methods for Partitioned Systolic Algorithms," *Proc. Application Specific Array Processors*, pp. 649-663, IEEE CS Press, Aug. 1992.

[18] D. Kulkarni, K. Kumar, A. Basu, and A. Paulraj, "Loop Partitioning for Distributed Memory Multiprocessors as Unimodular Transformations," *Proc. Int'l Conf. Supercomputing*, pp. 206-215, 1991.

[19] S.Y. Kung, *VLSI Processor Arrays*. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[20] D. Le, M. Ercegovac, T. Lang, and J. Moreno, "MAMACG: A Tool for Mapping Matrix Algorithms on to Mesh Connected Processor Arrays," *Proc. Application Specific Array Processors*, pp. 511-525, Aug. 1992.

[21] D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. Computers*, vol. 35, no.1, pp. 1-12, Jan. 1986.

[22] J.H. Moreno and T. Lang, "Matrix Computations on Systolic-Type Meshes: An Introduction to Multi-Mesh Graph (MMG) Method," *Computer*, vol. 23, no. 4, p. 23, Apr. 1990.

[23] J.H. Moreno, "Matrix Computations on Mesh Arrays," PhD thesis, Univ. of California, Los Angeles, June 1989.

[24] J.H. Moreno and M.E. Figueroa, "A Decoupled Access/Execute Processor for Matrix Algorithms: Architecture and Programming," *Proc. Application Specific Array Processors*, pp. 281-295, IEEE CS Press, 1991.

[25] J.J. Navarro, J.M. Llaberia, and M. Valero, "Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors," *Computer*, vol. 20, no. 7, pp. 77-89, July 1987.

[26] J.K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. 217-225, 1987.

[27] K.W. Przytula, "Medium Grain Parallel Architecture for Image and Signal Processing," *Parallel Architectures and Algorithms for Image Understanding*, V.K.P. Kumar, ed., pp. 95-119. Academic Press, 1991.

[28] W. Shang 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.

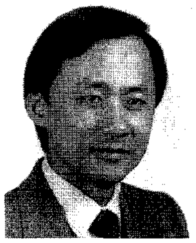
[29] A. Suarez, J.M. Llaberia, and A. Fernandez, "Scheduling Partitions in Systolic Algorithms," *Proc. Application Specific Array Processors*, pp. 619-633, IEEE CS Press, Aug. 1992.

[30] J. Symanski and K. Bromley, "Video Analysis Transputer Array (VATA) Processor," *Proc. SPIE Real-Time Signal Processing XI*, Aug. 1988.

[31] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 30-44, 1991.

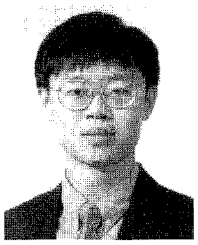
[32] X. Zhong and S. Rajopadhye, "Deriving Fully Efficient Systolic Arrays by Quasi-Linear Allocation Functions," *Proc. PARLE*, pp. 219-235, 1991.

Kumar Ganapathy received his BTech degree from the Indian Institute of Technology, Madras, India, his MS in electrical engineering from the University of Massachusetts at Amherst in 1990, and his PhD in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1994. Since 1994, he has been with Rockwell Semiconductor Systems, where he has played a key part in developing high-performance general-purpose DSP processors for high-speed communication and wireless applications. His research interests include DSP architectures, compilers, special-purpose and systolic architectures, processor validation, and low-power design techniques.



Benjamin W. Wah received his PhD degree in computer science from the University of California at 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 (1988-1989), as Fujitsu Visiting Chair Professor of Intelligence Engineering at the University of Tokyo (1992), and as McKay Visiting Professor of Electrical Engineering and Computer Science at the University of California at Berkeley (1994). In 1989, he was named a University Scholar of the University of Illinois. His current research interests are in the areas of parallel and distributed processing, knowledge engineering, and optimization.

Dr. Wah was editor-in-chief of the *IEEE Transactions on Knowledge and Data Engineering* from 1993-1996, and serves on the editorial boards of *Information Sciences*, *International Journal on Artificial Intelligence Tools*, and *Journal of VLSI Signal Processing*. He has chaired a number of international conferences and is currently serving the IEEE Computer Society as its treasurer and a member of its governing Board. He is the chair of the 1997 IEEE-CS Fellow Evaluation Committee. He is a fellow of the IEEE.



Chien-Wei Li graduated from National Taiwan University (BS, 1990, MS, 1992; both in computer science) and is currently a PhD student in the Computer Science Department and a research assistant at the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. His research interests are in computer architecture and compilers.