ON SYSTOLIC CONTRACTIONS OF PROGRAM GRAPHS

Weicheng Shen
Department of Electrical Engineering
University of New Hampshire
Durham, NH 03824-3591

A. Yavuz Oruç
Electrical, Computer, and Systems Engineering Department,
Rensselaer Polytechnic Institute,
Troy New York 12180-3590

ABSTRACT

One of the active areas in supercomputer research is concerned with mapping programs onto networks of processors. In this paper a variant of the mapping problem, namely, systolic contractions of program graphs are considered. The notion of time links is introduced to mechanize the contraction process, the timing of information flow between processors is modelled in terms of fundamental loop and path equations of delays, and optimized using linear programming.

Index Terms (Key Phrases): Graph contraction, fundamental loops of delays, program graph, processor graph, systolic array, time links.

Address for return of galley proofs:

A. Yavuz Oruç

Electrical, Computer, and Systems Engineering Department,

Rensselaer Polytechnic Institute,

Troy New York 12180-3590

Affiliation of authors and acknowledgement of financial support:

Weicheng Shen

Department of Electrical Engineering

University of New Hampshire

Durham, NH 03824-3591

A. Yavuz Oruç

Electrical, Computer, and Systems Engineering Department,

Rensselaer Polytechnic Institute,

Troy New York 12180-3590

This work was supported in part by the Electrical, Computer and System Engineering Department at Rensselaer Polytechnic Institute.

Captions of figures:

- Figure 1(a): The program graph for matrix/vector multiplication.
- Figure 1(b): The target processor graph for martix/vector multiplication.
- Figure 2: The appended program graph for matrix/vector multiplication.
- Figure 3: The program graph for a 3×3 convolution.
- Figure 4: The processor graph for a 3×3 convolution.
- Figure 5: The program graph for a palindrome recognizer.
- Figure 6: The processor graph for a palindrome recognizer.

1 Introduction

An active area in supercomputing research is concerned with designing systematic procedures for mapping algebraic computations onto networks of processors. Significant results have been reported in the literature, especially about mapping matrix computations onto a family of processor networks collectively referred to as *systolic arrays* [1-4]. These arrays prompted a considerable interest due to their regular structures, adjacency of interconnections between their processors and simplicity of their control.

Previous efforts on mapping algorithms onto systolic arrays deal with certain kinds of computations such as linear recurrences [3], matrix operations [2-4,7,8], 2-dimensional Fourier transforms and convolutions [10,11], finite impulse filtering [12] and string matching [10]. Attempts were made to establish mechanical procedures for mapping such computations to systolic arrays. For example, Weiser and Davis [10] formalized the data flows in systolic arrays by what is referred to as *wavefronts*. Through linear transformations of wavefronts, they described systolic executions of string matching and band matrix multiplication algorithms. Along the same direction, Johnson and Cohen [12] introduced delay operators to describe various time/space realizations of certain algebraic sums such as finite impulse filtering and discrete Fourier transforms. Capello and Steiglitz [14] used a geometric approach and modelled certain kinds of algebraic forms by *contours* of computation points in a three dimensional space with two spatial coordinates, and a time coordinate. Based on this geometric model they exhibited $(area) \times (time)^2$ near optimal systolic mappings of various algebraic forms. Near optimal mappings of various other computations were also presented in Li and Wah [3].

Though they work well for the applications they are designed for, the above efforts do not provide a generic, mechanical procedure for mapping computations onto systolic arrays. Such a procedure was recently reported by Ramakrishnan et. al [5] and Ramakrishnan and Varman [6] by using a family of labelled directed acyclic graphs which they refer to as *homogeneous program graphs*. A homogeneous graph denotes the algebraic dependencies in a block of statements in the usual way with the added labels signifying the streams of data

flow. The mechanical mapping procedure presented in Ramakrishnan et. al was based on a *diagonalization process* that makes use of the labels of edges to partition the computation vertices in the program graph into clusters and to assign each cluster to a unique processor in the target systolic array.

While this mapping procedure is quite powerful, it too has restrictions on both program and processor graphs which can be used in the mapping. The program graph must belong to a family of labelled graphs called *cube graphs,* and the processor graph must be a linear array, or have a rectangular or hexagonal geometry. Moreover, the procedure as described is not designed to handle mapping program graphs to processors arrays unless a processor array is specified a priori to the mapping. It is more appropriate to view such a mapping as a *contraction* since the target processor array is not known beforehand, and determined only after the mapping is completed. Contractions of program graphs are of theoretical as well as practical interest, since they are closely related to the question of how fast and cost-effectively we can execute program graphs by systolic arrays.

This paper provides an indepth characterization of systolic contractions of program graphs based on some key results reported earlier on space/time representations of array computations [14,15]. Unlike the earlier results, however, the focus here is on capturing the time delays over the links connecting the processing elements which are needed to guarantee the systolicity of the contraction in question. It is demonstrated that these delays obey three sets of equations which are, in a sense, analogous to fundamental loop and path equations in circuit theory.

The rest of the paper is organized as follows. Section 2 describes program graphs and contraction objectives. In Section 3 we introduce the notion of time links to characterize systolic contractions. In Section 4 we use this characterization to derive equations among the time delays over the links connecting the processing elements in a contracted program graph. We discuss how the relations among these sets of equations impact the values of time delays, and formulate the minimization of link delays between processors as a linear programming problem. In Section 5 we illustrate the contraction procedure by two examples. The paper

is concluded in Section 6.

2 The Computation Model and Mapping Objectives

We adopt the labelled directed acyclic graph model described in Ramakrishnan et. al [5] as the basis of representing programs. The following simple example conveys the idea behind the model. Let A be a 2×2 matrix, b be a 2-vector, and c be the product of A and b which is a 2-vector as well. The entries of c can be specified by the recurrence relation

$$c_k^{(i)} = c_k^{(i-1)} + a_{k,i}b_i, \;\; i = 1, 2, \;\; k = 1, 2,$$

with the initial conditions $c_1^0 = 0$ and $c_2^0 = 0$. Note that the product of A and b involves four multiply-add operations. The program graph describing the product Ab is depicted in Figure 1(a). The graph has four *computation* vertices each representing a multiply-add step as given in the above recurrence. The *in-degree* and *out-degree* for all the verticies is the same: $indegree = outdegree = 3$. The remaining vertices are referred to as the *source* and *sink* vertices, and represent, respectively, the operands and results of the program graph.

The main point to be made about the program graph in Figure 1(a) is that its computations have a regular pattern of data dependencies. The labelling scheme introduced in [5] exposes this regularity by associating each edge with a *label*. In this example, the horizontal edges are assigned the same label as they represent the entries of the resultant vector. Similarly, the vertical (and oblique) edges are assigned the same label as they represent the two input data streams. In this paper, it is assumed that all the program graphs are labelled unless otherwise stated.

A program graph may or may not correspond to the actual physical processor that executes the computation it represents. In fact, a direct execution of a program graph can often be very expensive, especially if it comprises a large number of computation vertices and/or edges. In such cases, it is best to map or *contract* the original program graph to a more feasible graph, i.e., a graph with fewer vertices and/or edges. Contraction of a program graph partitions its computation vertices into disjoint clusters and redefines the connectivity of the new graph based on the partition. More precisely, whenever, two computation vertices are

merged together, all the edges between the two are deleted, and all incoming and outgoing edges to these two vertices are reconnected to the resultant vertex without altering their orientations. The duplicate edges between the vertices in the resultant graph are also deleted unless their orientations or labels are different. A precise procedure for contracting program graphs will be stated in the next section, while Figure 1(b) depicts a contraction of the program graph of Figure 1(a). By merging the computation verticies along the vertical edges, a new graph with two vertices is obtained. This new graph also represents the same computation as does the original graph. However, there is a sharp contrast between the two: While each computation vertex in the original graph represents a single step, in the latter it corresponds to two steps. Moreover, each directed edge in the original program graph corresponds to a single operand, whereas in the latter it represents a *stream* of operands which move along that edge from one vertex to another. This is emphasized by the queues of inputs and outputs shown next to the sink and source vertices in Figure 1(b). We call a contracted program graph a *processor graph* to emphasize that the vertices in the resultant graph no longer represents the distinct steps of the original computation, rather they correspond to processors that execute several of these steps albeit one at a time.

3 Systolic Contractions

The preceding discussion illustrates that a program graph can be contracted to a processor graph by merging some of its vertices together. This contraction partitions the original computations into disjoint clusters and assigns each cluster to a separate processor. In addition, it must make sure that the operands for each computation are available to the processor to which it is assigned. In general, operands can be provided to processors in many ways. The options range from simultaneously fanning-in all the relavant operands into the processor in question to forming streams of operands which move along directed paths of processors. The contractions considered in this paper belong to the latter category, and are referred to as *systolic contractions*. More precisely, we state the following definition.

Definition 1 *The contraction of a program graph is called systolic if the resulting graph has the following properties:*

(a) The operands within the same stream move at the same speed and arrive at each processor on its way in sequence, i.e., with fan-in = 1 and leave it in sequence, i.e., with fan-out = 1 , one operand at a time,

(b) All of the operands, which form a computation in the original program graph, arrive simultaneously at the processor to which the computation is assigned,

(c) Each processor executes the same operation in every clock cycle as does every other processor.

Intuitively, Case (a) means that data travel in and out of processors along streams, and the streams are not permitted to divide into substreams. It also implies that the rate of arrival of operands within the same stream at a processor is deterministic and uniform. Case (b) implies that each processor must receive its operands forming a computation simultaneously, and no memory is available even for temporarily storing them. Note, however, this does not mean that operands cannot be recirculated through a delay-line back onto a processor. In fact, this is exactly what we shall do when an operand is needed by a processor several times before it completes its computation. This does not contradict Case (b), as all the operands will arrive at a processor at the same time, albeit some will arrive more than once. Case (c) requires that every processor execute the same operation in every clock cycle, and cannot switch between different computations. This is probably the most unreasonable of all the assumptions. However, it enforces homegeneity and simplicity, which are considered to be salient features of systolic arrays.

To further characterize systolic contractions, we introduce the notion of *time links* which is analogous to the space/time representation described by Miranker and Winker [15], and the diagonalization notion of Ramakrishnan et. al [5]. Time links are appended to the program graph to identify the computation vertices to be merged together without violating the data dependencies. A program graph appended with time links is called an *appended*

program graph. When a time link joins two computation vertices, they are assigned to the same processor, and the vertex from which it moves away is computed first. If a computation vertex is not attached to a time link then it is mapped to a processor vertex by itself.

An oriented path solely consisting of time links is called a *time path*. Only those computation vertices that fall on a given time path are mapped to the same processing element in the contracted graph. Notice that a computation vertex cannot be on more than one time path as this would imply that it be mapped to at least two processor vertices which is not permitted. We also shall not permit edges with different labels to be on the same time path as this would require the recirculation of different streams around the same processor which would then require switching between different streams of operands, thereby violating Case (c) of Definition 1.

Apart from these provisions, we provide a further characterization for systolic contractions by a notion called the *juxtaposition* of time paths as follows. Suppose that a set of time links are appended to a program graph, and all the edges, except those with one label, say β, are deleted from it. Call this graph, the *extraction* of the appended program graph with respect to label β. As an example, an appended version of the program graph in Figure 1(a), and its extraction with respect to label s_3 are depicted in Figure 2. The dashed lines indicate the time links.

Definition 2 *A pair of time paths in an extraction of an appended program graph are said to be juxtaposed if and only if the following are true:*

(a) All the edges with the same label and moving away from one time path arrive at the other.

(b) Edges with the same label and moving away from consecutive vertices on one time path arrive at consecutive vertices on the other with the same orientation. ||

We can now state the main result of this section.

Theorem 1 *If a program graph is systolically contractible to a processor graph then every pair of time paths connected by edges with the same label in each extraction of the appended program graph with respect to that label must be in juxtaposition.*

Proof: Suppose that a pair of time paths are not in a juxtaposition in a given extraction of the program graph in question. Then either of the two cases in the above definition must be false. Suppose that Case (a) does not hold. Then there must exist at least three time paths, p, q and r corresponding respectively to three processors p, q and r with vertices u and v on p, w on q, and z on r such that there is an edge with label β connecting vertex u to vertex w, and an edge also with label β connecting vertex v to vertex z. Moreover, since u and v are to be executed on processor p, w on processor q, and z on processor r, processor p must have two edges with the same label, one connecting to processor q, and the other connecting to processor r. But this means that the operand stream corresponding to label β is split into two substreams as it leaves processor p, thereby contradicting Case (a) of Definition 1.

Suppose Case (b) does not hold. Then either consecutive vertices on one time path are not mapped onto consecutive vertices on the other, or they are, but their orientations are not preserved. Consider the first case, and let the edges between some two time paths have the label β. After the program graph is contracted, the operands associated with these edges will be assigned to the same stream. Now, by Case (a) of Definition 1, all operands within a given stream must move at the same speed. Therefore, if consecutive vertices in one time path are not connected to consecutive vertices in the other, then there exists a processor at which not all the prerequisite operands can be assembled simultaneously, or there exist at least two processors operating at different speeds. These, respectively, violate the last two cases of Definition 1.

Now suppose that consecutive vertices are mapped to consecutive vertices, but their orientations are not preserved. Then one of the time paths must have two consecutive vertices u and v in that order, and the other two consecutive vertices w and z in that order such that there is an edge from u to z, and an edge from v to w. But this means that, either for some computation, not all of the operands of one of the processors arrive at it simultaneously, or the processors associated with the two time paths are not operating at the same speed. These, again, contradict the last two cases of Definition 1. ||

This theorem can be used to determine whether a contraction of a program graph satisfies the systolicity property as characterized by Definition 1. It does not, however, lead to a systolic contraction by itself. This should be expected since the juxtaposition of time paths makes no explicit reference to how the operands are grouped into streams, how fast the operands within each stream must move, or how much delay must be assigned to the edges between processors, etc. We discuss these issues in the subsequent sections.

4 Delays Between Processors

In an appended program graph, each edge has a delay constant associated with it. In general, these delay constants can be made arbitrarily large as long as one assumes that the procesors can hold their operands temporarily until all are available. Here the penalty is extra storage and longer execution time, and as we have stated in the previos section, systolic contractions do not allow temporary storage and require that all operands of each processor become available within the same clock period. We can use this fact as a basis to determine the delays for the edges between the processors. Here the key observation is that the delays assigned to the directed edges within a loop or cycle in the appended program graph sum to zero. That is,

Theorem 2 *If an appended program graph is systolically contractible then the algebraic sum of the delays of the edges in any of its oriented loops must be zero. More precisely, if the set of edges* $\{e_1, e_2, \ldots, e_k\}$ *forms a loop then* $\Sigma_{i=1}^{k} b_i d_i = 0$ *where* $b_i = 1$ *if the orientation of* e_i *agrees with the orientation of the loop, and* $b_i = -1$ *if it disagrees with it.*

Proof: It should be clear that if there is a directed edge in the appended program graph in question with delay d and oriented from vertex u towards vertex v, then the execution of vertex v must follow the execution of vertex v by d clock periods. Now consider an arbitrary loop in the graph containing two verticies x and y among others. Then there exist two distinct paths connecting these two verticies. Let these paths be associated with the sequences of edges, $e_1, e_2, \ldots, e_k$ and $e_{k+1}, e_{k+2}, \ldots, e_n$ where n is the total number of edges in the loop. By Case (b) of Definition 1, the sums of delays along these two paths must be

identical. Thus, if t_x denotes the time when x is executed, then summing the delays two different ways, we have $t_x+\Sigma_{i=1}^{k} b_i d_i = t_x+\Sigma_{i=k+1}^{n} b'_i d_i$ where $b'_i = -b_i$ for $i = k+1, k+2, \ldots, n$. It follows that $\Sigma_{i=1}^{n} b_i d_i = 0$, and hence the assertion. ||

The reader should note the analogy between this result and Kirchoff voltage law in circuit theory. There, a fundamental loop matrix is used to characterize all the independent loops in an electric circuit. Here, we shall use a similar matrix to characterize the fundamental loops of delays in an appended graph. The objective will then be to compute the delays in these loops.

The fundamental loops of any graph with n edges and m vertices can be represented compactly by an $(n-m-1) \times n$ matrix $\mathbf{A} = [a_{i,j}]$, where the rows correspond to loops and columns to the edges, and

(i) $a_{i,j} = 1$ if edge j is in loop i, and they have the same orientation,

(ii) $a_{i,j} = -1$ if edge j is in loop i and they have opposite orientation,

(iii) $a_{i,j} = 0$ if edge j is not in loop i.

The rows of $\mathbf{A}$ are independent and hence $\mathbf{A}$ has rank $n-m+1$. In circuit theory, these $n-m+1$ rows correspond to $n-m+1$ voltage equations in a network with m vertices and n edges and without dependent voltage or current sources. This is compactly represented by $\mathbf{A} \cdot \mathbf{v} = \mathbf{0}$ where $\mathbf{v}$ is a vector of n unknown voltages. In our case, the unknown variables are delays associated with the edges of an appended program graph. Thus a similar relation holds for the unknown delays, i.e., $\mathbf{A} \cdot \mathbf{d} = \mathbf{0}$ where the rows of $\mathbf{A}$ correspond to fundamental loops of delays, and $\mathbf{d}$ is a vector of n unknown delays.

In addition to these equations, we derive two more sets of equations from Definition 1. First, since each processor executes at a fixed speed, the time between the executions of consecutive operations assigned to the same processor must be the same. This requires that the delays associated with the edges on each time path in the appended program graph be equal. Thus, if the number of processors into which two or more computations are merged is r, and the number edges absorbed by processor i is n_i, then the first case induces an equivalence relation which partitions the delays into r sets where the cardinality of the ith

set is n_i. Therefore, the *ith* set corresponds to $n_i - 1$ linearly independent equations, and all of the sets combined correspond to $n_1 + n_2 \cdots + n_r - r$ linearly independent equations which will be referred to as *time-path equations*.

Definition 1 also requires that the delays associated with the edges with the same label and which fall between two processors also be equal. This is due to the fact that when the program graph is contracted, such edges will be merged to a single link, and must, therefore, be given the same delay. Finally, the delays associated with the edges which are formed into streams at the end of the contraction must also be equal by Case (a) of Definition 1. Thus, if the number of streams, which are formed as a result of the contractions, is q, and the *ith* stream contains n'_i edges, then, we have a set of $n'_1 + n'_2 \cdots + n'_q - q$ linearly independent equations which will be referred to as *stream equations*.

The fundamental loop, time-path, and stream equations along with the constraint that each $d_i \geq 1$ form the basis for determining the delays in an appended program graph. It should be noticed that, since each edge in an appended program graph falls on either a time path or a stream, there are $n - r - q$ time-path and stream equations combined together, and they constitute a linearly independent set. However, the loop equations when combined together with these two sets of equations need not form a linearly independent set.

If the number of linearly independent equations contained in the three sets is more than n then the solution is trivial, i.e., $d_i = 0$, for all $i = 1, 2, \ldots, n$. Given that the delays must be non-zero, this case implies that a systolic contraction is impossible. On the other hand, if the number of linearly independent equations is n, then the solution of delays is unique, and can easily be determined.

Finally, if the number of linearly independent equations is less than n then there are infinitely many solutions. Among these, we are interested in solutions which minimize certain linear combinations of delays which are of the form $c_1 d_1 + c_2 d_2 \cdots + c_n d_n$. This allows a variety of minimization objectives including that which minimizes the propagation delays over critical paths. Such paths determine the computation time of the program graph in question. The minimization problem can be formulated as a linear programming problem as

follows.

Minimize $c_1x_1 + c_2x_2 \cdots c_nx_n$ *subject to the constraints* $\mathbf{Md}' = \alpha$ *and* $\mathbf{d}' \geq \mathbf{0}$.

Here $\mathbf{M}$ is a known $p \times n$ matrix representing the p linearly independent equations contained in the above three sets of equations, $\mathbf{d}'$ is a translation of the unknown n-vector $\mathbf{d}$ of delays, and α is a known m-vector. This problem can be solved by the two-phase method [17] which will not be detailed here. We also remark that a similar minimization formulation was reported in Cytron [16] for the delays between the executions of statements within nested loops in a program.

5 Contraction Procedure and Timing

The results established in the preceding sections can be combined into the following contraction procedure.

(1) Append time edges to the program graph in question so that the computations which need to be scheduled for the same processor fall onto the same time path when they are juxtaposed. This step serves to partition the computations of the program graph and to map each cluster to a separate processor.

(2) Contract the appended program graph by replacing all the vertices which fall on the same time path by a single processor vertex, and merging all edges with the same label and orientation between all pairs of time paths into a single edge.

(3) Determine the delays associated with the edges between the processors in the contracted graph, and based on these delays, form the operand streams and determine their timing.

We illustrate these steps by two examples. First consider the mapping procedure for computing a 3×3 convolution. The computations are

$$\begin{aligned} y(0) &= x(0)h(0) \\ y(1) &= x(0)h(1) + x(1)h(0) \end{aligned}$$

$$
\begin{aligned}
y(2) &= x(0)h(2) + x(1)h(1) + x(2)h(0) \\
y(3) &= x(1)h(2) + x(2)h(1) \\
y(4) &= x(2)h(2)
\end{aligned}
$$

This set of equations can be represented by the program graph shown Figure 3 where each vertex represents a multiply-add operation. In this case, time edges can be appended in anyone of three ways: We can replace the edges corresponding to the h operands by time links, replace those corresponding to the x operands, or replace those corresponding to the y operands. The first two of these result in a processor graph with 3 processors. The last one leads to a processor graph with 5 processors as shown in Figure 4. It can easily be shown that all of these contractions obey the juxtaposition principle stated earlier in Section 3.

To determine the delays for the links of the processor graph which is obtained by the last contraction, we first note that the graph in Figure 3 has $n - m + 1 = 16 - 9 + 1 = 8$ loop equations, 1 time-path equations, and 10 stream equations. As claimed earlier in a more general setting, the 11 time-path and stream equations form a linearly independent set. In this case, we can add one fundamental loop equation to this set, such as, $d_1 - d_3 + d_2 = 0$, and still maintain the linear independence among the 12 equations. Furthermore, it can be shown that any other linear equation involving the 16 delays can be obtained from these twelve equations.

Given 12 equations and 16 delays to be determined, there are infinitely many solutions. One solution of interest is one which minimizes the sum of delays, $d_1 + d_2 + d_3$ subject to $d_1 - d_2 + d_3 = 0, d_1, d_2, d_3 \geq 1$. The solution is $d_1 = 1, d_2 = 2, d_3 = 1$. The delays associated with the remaining edges in Figure 4 can all be determined directly from d_1, d_2, and d_3.

As assigned, these delays make sure that all operands arrive at each processor simultaneously provided that the arrivals of the x and h operands are properly fixed. We synchronize the x and h operand streams as shown in Figure 4. The skewing of the x-stream to the left of the h-stream by 2 clock cycles along with the determined values of d_1, d_2, and d_3 guarantees that the operands arrive at each processor simultaneously. Note that, unlike the x and h operands, each of the y operands is separately fanned-in and recirculated back into

the corresponding processor.

As a second example, we consider a systolic solution of a palindrome recognizer. This problem was considered earlier by Cole [18], and Leiserson and Saxe [13] using processing elements which can switch between operations. The systolic solution provided here is based on processing elements with no such decision capability.

Let $B_n = b_1 b_2 b_3 \ldots b_n$ be a string of n symbols. B_n is said to be a palindrome if $b_i = b_{n-i+1}; 1 \leq i \leq \lceil n/2 \rceil$. This can be expressed recursively as

$E_n^i = E_n^{i-1} \wedge (b_i \odot b_{n-i+1}); 1 \leq i \leq \lceil n/2 \rceil$, $E_n^0 = 1$.

where $\odot$ denotes the exclusive-nor function.

The graph in Figure 5 depicts a program which recognizes whether any string of up to 7 symbols is a palindrome. Each of the vertices in the graph represents the logical computation on its incoming edges as given above. This program graph, as in the earlier example, can be contracted in anyone of three ways: We can replace the edges with label s_1 by time links, replace those with label s_2, or replace those with label s_3. The first and last contractions each result in a processor graph with 7 processors. The second contraction leads to a processor graph with 4 processors as shown in Figure 6. The delays for the links are determined by the triangular loops which require that the operands on the E-stream move twice as fast as those on the a_i-stream. The minimum delay values which satisfy this constraint are 2 clock cycles between consecutive operands on the a_i-stream, and 1 clock cycle between consecutive operands on the E-stream. As in the earlier example, each of the vertical operands is separately fanned-in and recirculated back to the corresponding processor.

6 Concluding Remarks

In this paper, a procedure has been described for contracting program graphs into processor arrays. The contraction procedure has been demonstrated through the linear convolution of two sequences, and a palindrome recognizer. Three sets of delay equations, namely, time-path, stream, and loop equations, have been derived to guarantee the systolicity of

contractions. This means that operands move at a fixed speed along streams, and any set of operands forming a computation at a given processor arrive at it simultaneously. This eliminates the need for storage when processors perform their computations.

While the results of this paper apply primarily to systolic contractions, they can be extended to other types of graph contractions by relaxing the timing constraints imposed by delay equations. Such contractions can, in principle, lead to faster executions of program graphs since they may do away with the delays imposed on the links between the processors by systolic contractions. Clearly, here, storage is traded for time since extra storage may be necessary to temporarily hold some operands until a processor acquires all of its operands for a computation. A further investigation of this tradeoff may provide a better understanding of the role that systolic arrays play in parallel computing.

1. H. T. Kung, "Why systolic arhitectures," *IEEE Computer*, Vol. 15, pp. 37-46, Jan. 1982.

2. S. Y. Kung, "On supercomputing with systolic/wavefront array processors," *Proc. IEEE*, Vol. 72, pp. 867-884, July 1984.

3. G. J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput*, Vol. C-34, pp. 66-77, Jan. 1985.

4. H. T. Kung, and C. E. Leiserson, "Systolic arrays for VLSI," In Proc. *SIAM* Sparse Matrix Proc., pp. 256-282, 1979.

5. I. V. Ramakrishnan, D. S. Fussel, and A. Silberchatz, "Mapping homogeneous graphs on linear arrays," *IEEE Trans. Comput.*, Vol. C-35, pp. 189-209, March 1986.

6. I. V. Ramakrishnan and P. J. Varman, "On mapping cube graphs on VLSI arrays," 4th Int. Conf. Foundations Software Technol., Lecture Notes in Computer Science, Vol. 181, Berlin, Springer Verlag, Dec. 1984.

7. I. V. Ramakrishnan, and P. J. Varman, "Modular matrix multiplication on a linear array," *IEEE Trans Comput.*, Vol. C-33, pp. 952-958, Nov. 1984.

8. P. J. Varman and I. V. Ramakrishnan, "Synthesis of an optimal family of matrix multiplication algorithms on linear arrays," *IEEE Trans Comput.*, Vol. C-35, pp. 989-996, Nov. 1986.

9. D. I. Moldovan, " On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, Vol. No. 1, pp. 113-120, 1983.

10. U. Weiser, and A. Davis, "A wavefront notation tool for VLSI array design," In VLSI Sys. and Comput., H. T. et. al, Eds., Rockville, MD, pp. 226-234, 1981.

11. W. C. Shen, "On Mapping Algorithms onto Processor Arrays," Ph.D. Disser., ECSE Department, RPI, Dec. 1987.

12. L. Johnson and D. Cohen, "A mathematical approach to modelling the flow of data and control in computational networks," In VLSI Sys.and Comput., H. T. Kung et. al, Eds., Rockville, MD, pp. 213-225, 1981.

13. C. E. Leiserson, and J. B. Saxe, "Optimizing synchronous systems," In proc. 22nd Annu. Symp. Found. Comp Sci., pp. 23-36, Oct. 1981.

14. P. Capello, and K. Stieglitz, "Unifying VLSI array designs with geometric transformations," In proc., Intl. Conf. Parall. Proc., pp. 448-457, Aug. 1983.

15. W. L. Miranker and A. Winkler, 'Spacetime representation of computational strutures," *Computing*, Vol. C-32, No. 2, pp. 93-114, 1984.

16. R. Cytron, "Doacross: Beyond vectorization for multiprocessors (Extended Abstract)," In proc. Intl. Conf. Parall. Proc., pp. 836-844, 1986.

17. D. G. Luenberger, *Linear and Nonlinear Programming,* Reading, MA., Addison-Wesley Pub., 1984.

18. S. N. Cole "Real-time computation by n-dimensional iterative arrays of finite-state machines," *IEEE Trans. Comput.*, Vol. C-18, pp. 349-365, Apr. 1969.