I/O Behavior of Systolic Arrays

Sanjay Rajopadhye

August 11, 1988
CIS-TR-87-16A*

Department of Computer and Information Science
University of Oregon

*This report is a revision of CIS-TR-87-16.

I/O Behavior of Systolic Arrays*

Sanjay V. Rajopadhye
Computer Science Department
University of Oregon
Eugene, Or 97403
sanjay@cs.uoregon.edu

August 10, 1988

Abstract

The problem of loading (unloading) data into (from) a systolic array is addressed, in the same context as the synthesis problem. The primary constraint is that all I/O must occur at boundary processors. It is shown that a simple linear condition imposed on the allocation function is necessary and sufficient to satisfy this constraint. This may involve augmenting the domain and introducing new dependencies into the algorithm. We describe how the original specification is to be augmented, how the new recurrence is to be analyzed, and finally how the target architecture and the control signals that govern the I/O are to be derived within a single unified framework.

1 Introduction

Systolic and wavefront arrays [7,8,9,11] are an important class of parallel architectures that have shown great promise for exploiting the advances in VLSI technology. They consist of *regular* interconnections of a *large* number of very *simple* processors, and are typically used as back-end processors for computation-intensive processing. A number of researchers have studied the problem of *automatically* deriving systolic implementations for a given algorithm. The results due to Moldovan,

*To apperar in "VLSI Signal Processing," 1988

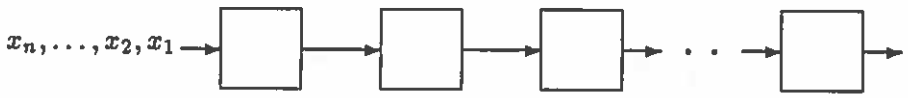


Figure 1: Naive Systolic Sorter

Fortes, Quinton and others [2,3,12,13,10] (see [15] for an extensive survey) are now well known and a systematic methodology for systolic array synthesis has emerged. The technique consists of analyzing the dependencies of an algorithm and mapping the problem domain to a *space-time* domain, based on the results of the analysis. Much of the earlier work was applicable to algorithms that had what is called a *uniform* dependency structure (such algorithms have been characterized by Uniform Recurrence Equations, UREs). Recently, we have shown how such architectures can be synthesized when the initial specification is a more general class of algorithms (as defined by Affine Recurrence Equations, AREs) [15,17]. We have also described how to automatically derive *control signals* and *boundary-processor functionality* in the array [14,16].

While all these results have been encouraging, the effort so far has concentrated on the problem of *mapping* the initial specification to a regular and local target architecture. If systolic arrays are to be a viable option in the parallel computing milieu, it is also important to be able to *systematically derive* how they should be loaded and unloaded efficiently. As an illustration, consider the linear systolic array shown in Figure 1. It consists of a cascade of simple processors, each one having a single input and output, and an internal accumulator (initialized to the smallest possible value, $-\infty$). At each clock tick, the processor receives an input value, compares it with the contents of the accumulator, and sends the smaller of the two on its output lines, while storing the larger one in the accumulator. Thus, by the time n numbers have streamed past the processor, it has identified the largest number and the output stream contains $n-1$ numbers. As a result, an array of n such processors is capable of sorting n numbers. If the inputs are available at leftmost processor at $t = 0$, they are completely scanned by the first processor at $t = n$. At this time instant, the second processor has scanned only $n-2$ numbers.* The second processor thus takes one more clock tick to finish its computation (i.e., at $t = n+1$), and similarly the k-th processor completes its computation at $t = n+k-1$. The input is sorted when the n-th processor finishes, i.e., at $t = 2n-1$.

*Note that while the first processor begins its computation at $t = 0$, the second processor sees $-\infty$ on its input at $t = 1$, and only at $t = 2$ does it see its first "meaningful" input.

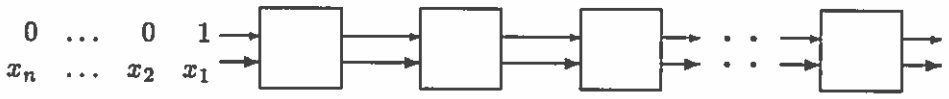


Figure 2: Sorting Architecture with Control Signals

Given such an architecture, how should the input data be loaded into the array and how should the results be unloaded? A naive approach would be to simply "pad" the input with n copies of the largest possible value, ∞. As a result, the numbers will be "pushed out" in exactly the sorted order. Since there are now $2n$ input values, the k-th processor finishes its computation at $t = 2n + k - 1$ and thus the n-th processor completes its computation at $t = 3n - 1$. At this time instant, all the accumulators contain ∞ and the sorted numbers have emerged from the rightmost processor. Thus the total computation time of the architecture is $3n - 1$. If the architecture is to be reused for sorting another set of data, the accumulators must now be initialized to $-\infty$, presumably by a global control signal that is broadcast to all the processors. Because of this, a new computation cannot be started before $t = 3n$.

On the other hand, consider the architecture shown in Figure 2 which is identical to the earlier one, except that the processors now have a control signal that is propagated through the array. The control signal indicates to the processor that the current input is the first element of a new set of data. At every time instant the processor reads the control value and, if it is one it unconditionally sends the accumulator contents on its output and loads the accumulator from the input value. Otherwise, the processor performs the normal computation described earlier. In either case, the control value is delayed by one time unit (by storing it in a local buffer and making it available at the output only at the next clock tick). As a result, the array needs no initialization. A straightforward analysis reveals that while the total computation time is still $3n$ (actually $3n - 2$), a new set of data can be input into the processor at time $t = n$. As a result we are able to *overlap* a number of "process-level" computations, thus improving the overall performance of the architecture.

Why does the naive approach of padding the input stream with ∞'s cause the data to be output correctly? Why does the introduction of control signals into the naive architecture provide such a dramatic improvement? Is there a general principle governing such behavior? And most importantly, can such a principle be unified into the techniques for automatic synthesis, in such a manner that the architectures that are synthesized will have optimal input-output performance? These are the questions that we address in this paper. Our main thesis is as follows.

We know that the standard synthesis technique consists of *mapping* a problem domain to a target (*processor-time*) domain by using affine transformations. We also know that this mapping must have an *inverse* and thus, if the processor and the time instant is known, the index point in the original domain can be uniquely determined. Given this basic paradigm, we impose the constraint that all input and output must occur only at the boundary processors.* As a result, we must investigate the behavior of the boundary processors as a function of time, i.e., a set of points in the processor-time domain. These points lie in a straight line (a plane for two-dimensional arrays) and are the *image* (under the affine transformation) of another line (or plane) in the *problem domain*. If this line is not a subset of the problem domain we must *augment* the domain to include these points. This may entail a re-analysis of the whole problem, which may now no longer have uniform dependencies. In the remainder of this paper, we describe how the augmented problem specification is obtained, how the new (non-uniform) dependencies are *pipelined*, and finally how the target architecture is generated (including automatic derivation of the control).

Some early work on the I/O problem has been reported by Engstrom and Cappello [1] where they describe how to "extend" the data dependencies in a space-time diagram. However, the primary focus of the work reported there is on *specifying* systolic designs, and they do not give a formal treatment of how the dependencies are to be extended. Jagadish et. al. [5] have independently developed a process called *expansion* of the index-space that is similar to our approach in that, they also extend the index space in a direction that is parallel to the direction of the *iteration space* (analogous to our allocation function). Their approach is applicable to their Regular Iterative Algorithms, which have a *uniform* dependency structure and there has been no attempt to indicate how this process may be automated, what is to be done in the case of a choice of possible directions along which the expansion may be prformed, etc. The remainder of this paper is organized as follows. We first describe the overall synthesis methodology and illustrate it by describing how the architecture of Figure 1 may be derived. In Section 3 we show that if the I/O is to be performed only by the boundary processors, then it should be computed on *parallel* boundaries of the initial specification. We also give a set of conditions under which the I/O is performed by the boundary processors. These results are illustrated for the sorting array in Section 4.

*In a linear array, we consider only the two end processors as the boundaries. In a planar array the boundary processors are the *edges* of the array.

2 Outline of the Methodology

Typically, the initial algorithm is defined by the computation of a function at all points in an *index-space* (*viz* the integer lattice points in a subset of Euclidean space). An example of such an algorithm is a simple (loop) program. The range of values over which the loop indices are permitted to vary define exactly the index-space (which is n-dimensional if there are n nested loops); the body of the loop represents the function which is computed at all the points in this domain, and thus serves to define the granularity of the processor. The synthesis problem then reduces to mapping the original index-space to a *space-time* domain (i.e., assigning to each point in the original domain a *place* and a *time*). The two parts of the mapping function are referred to as the *allocation function* (for the space component) and *timing function* (time component) respectively. Such a mapping must satisfy the following constraints.

- The mapping must be bijective i.e., two distinct points in the index-space should be mapped to two distinct points in the space-time domain. If it were not so, the computations from two distinct points in the problem domain would be scheduled on the same processor at the same time, giving rise to a *conflict*.
- The data dependencies of the original algorithm must be rendered spatially and temporally local since systolic arrays have nearest neighbor interconnections (spatial locality) and a finite memory in each processor.*
- These transformed dependencies must be *uniform* over the whole space-time domain, since the processors in a systolic array are *identical* and have similar interconnections independent of their physical location in the array. The importance of this requirement has been demonstrated elsewhere [15].
- The time component must preserve the dependencies of the original index-space, *i.e.*, in order to schedule the computation at any point, *all* its arguments must first be evaluated.

Under this framework it is now common to view the initial algorithm as a *recurrence equation* i.e., an equation defined as

$$f(p) = g(f(q_1), f(q_2), \ldots, f(q_k))$$

*Note that the finite memory mandates temporal locality, since the value used by any processor must have been produced by its neighbor only a finite number of "clock-ticks" ago.

where g represents the loop body and p and $q_1 \ldots q_k$ are n-dimensional points. The domain is defined separately and is usually a convex hull. In much of the earlier work, the recurrence was restricted to be a *Uniform Recurrence Equation* (URE). In UREs the difference $p - q_i$ between a point p and any other point q_i on which it depends, was required to be a *constant vector* independent of p. It is easy to show that if *affine transformations* are applied to such an index-space, the transformation yields another index-space that also has uniform dependencies. This means that the third constraint mentioned above can be easily satisfied by appropriate *affine* timing and allocation functions. Hence, by choosing appropriate affine timing and allocation functions, it can be guaranteed that the processors will have identical interconnections. A number of earlier researchers have developed necessary and sufficient conditions for the existence of such timing and allocation functions . They also showed that the problem of determining such functions reduces to a constraint optimization problem with a *constant* number of constraints.

We have argued elsewhere [17] that requiring the dependencies to be uniform at the *initial specification* level is unduly restrictive and proposed a more generalized class of recurrences called *Affine Recurrence Equations* (AREs). In AREs each of the q_i's are *affine* functions of p, i.e., $q_i = \mathrm{A}_i p + \mathrm{b}_i$ where A_i is an $n \times n$ matrix and b_i is an $n \times 1$ vector. We have shown that under affine transformations the dependencies of AREs remained affine (and hence did not represent a systolic array). It was therefore necessary to *explicitly pipeline* the dependencies to render them uniform. A similar idea was discussed by Fortes and Moldovan [3].

The basic idea behind pipelining is as follows. In an ARE, the computation of f at any point p requires the value of $f(q)$ as one of its arguments (where $q = \mathrm{A}p + \mathrm{b}$). Clearly, the difference $p - q$ is not a constant vector, but depends on p. To render the dependency uniform let us first try to find another point p' that also requires the value of $f(q)$. We want p' to be "close to" p in the sense that $p' - p$ is a constant vector independent of p (say ρ). If such a point exists, and somehow it can be correctly scheduled (i.e., all its arguments can be made available at p') then we may also schedule p as follows. While computing $f(p)$, instead of obtaining the argument $f(q)$ from the point q we obtain it from p' (which is just a constant vector away and which has just used the value in its own computation). To do this we introduce an additional *pipelining function* $f'(p)$ at all points in the domain. This function merely passes its arguments on unchanged. Then the value of $f(q)$ is available at point p from p', i.e., $p + \rho$ (which in turn gets it from $p + 2\rho$ and so on). Carrying this argument to its inductive conclusion, the pipelining function has the following form.

$$f'(p) = \begin{cases} f(p+\rho') & \text{if } \pi^{\mathrm{T}} p = \theta \\ f'(p+\rho) & \text{otherwise} \end{cases}$$

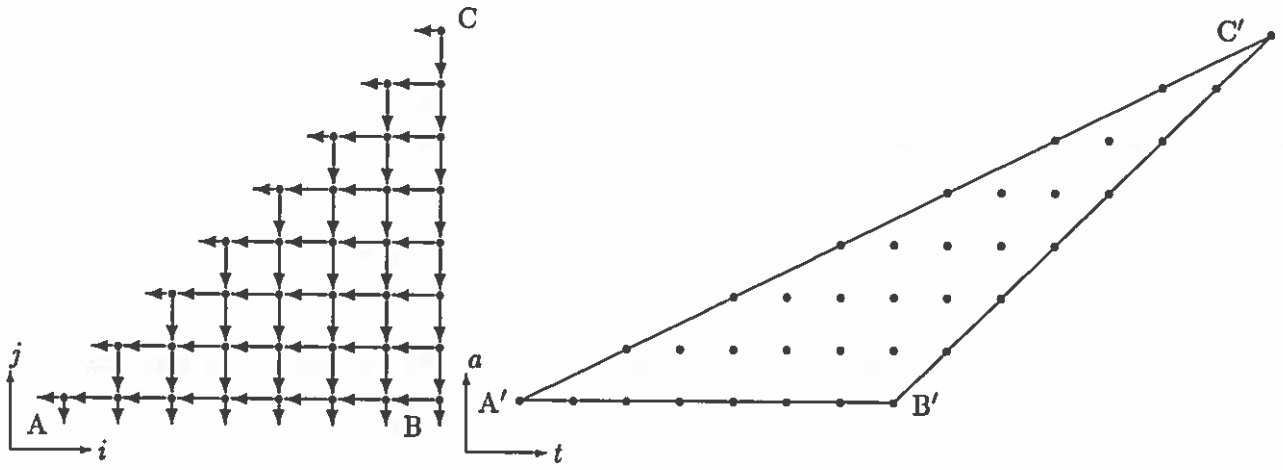
The expression $\pi^{\mathrm{T}} p = \theta$ is a "termination condition" to test if the point p is on a domain boundary (the basis case for the inductive argument above), and the new (uniform) dependency that has been introduced is ρ. The pipelining function f' does not impose any additional "computational" requirements on the processor since at the domain boundary its value is simply the same as f, and at internal points it is the same as f' at a neighboring point. It is thus possible to transform the dependencies of an ARE into uniform ones. The conditions under which such a *source-to source* transformation can be performed are discussed elsewhere [17] and a constructive method for obtaining the new dependencies is also given. In particular, it has been shown that pipelining may be performed if the original affine dependency [A, b] satisfies certain properties (which can be verified by straightforward linear algebra), and ρ corresponds to the null space of the matrix A. Once the dependencies have been made uniform, we may use existing techniques of determining appropriate timing and allocation functions to derive the target array.

2.1 Derivation of the Sorting Architecture

We shall now briefly illustrate the basic approach by describing how the architecture of Figure 1 may be systematically derived. The initial initial specification (Figure 3a) corresponds to Knuth's network for the bubble sort algorithm [6].* This recurrence is already in the form of a URE, where the computation performed at each point $[i, j]$ in the triangular domain consists of evaluating two functions f_1 and f_2 of two arguments that are obtained from points $[i-1, j]$ and $[i, j-1]$. It is described by the recurrence in Equation 1. The input values are required by the points $[i, 1]$, i.e., at the horizontal boundary, AB, of the domain, and the output is computed at points $[n, j]$, the vertical boundary, BC.

$$\begin{aligned} f_1(i,j) &= \begin{cases} \text{dont care} & \text{if } i = j \\ \min(f_2(i-1,j), f_1(i,j-1)) & \text{otherwise} \end{cases} \\ f_2(i,j) &= \begin{cases} f_1(i-1,j) & \text{if } i = j \\ \max(f_2(i-1,j), f_1(i,j-1)) & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

*Note that the arrows in the figure represent *dependencies* and are therefore opposite to direction of the data flow.



a: Initial Domain b: Target Domain

Figure 3: Dependencies of the Bubble Sort Algorithm

To derive a systolic architecture from this specification, we must determine *affine* functions $t(i,j)$ and $a(i,j)$ (the *timing* and *allocation* functions), and it can be easitly shown that $t(i,j) = i+j-1$ and $a(i,j) = j$ respectively satisfy the constraints of causality and locality described above. Thus, $t(i,j)$ is a family of straight lines with a 45-degree slope (as shown in Figure 3a) and $a(i,j)$ is simply a projection of the domain on the j-axis. This affine transformation is thus described by the following equation.

$$\begin{bmatrix} x \\ t \end{bmatrix} \equiv \Lambda \begin{bmatrix} i \\ j \end{bmatrix} + \alpha = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Under this mapping, the *image* of the domain ABC is the triangle A′B′C′ as shown in Figure 3b, and the mapping *completely* defines the sorting architecture of Figure 1. For example, since $a(i,j) = a(i-1,j) = j$, we know that the j-th processor gets its f_2 value from the j-th processor itself (i.e., from an internal accumulator), and its f_1 value from the neighboring processor (since $a(i,j-1) = j-1$). Moreover, for any point $[x,t]$ in the processor-time domain, the corresponding problem domain point $[i,j]$ can be easily determined by the inverse mapping.

$$\begin{bmatrix} i \\ j \end{bmatrix} = \Lambda^{-1}\left(\begin{bmatrix} x \\ t \end{bmatrix} - \alpha\right) = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} x \\ t \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

3 I/O Constraints and their Relation to Synthesis

As mentioned above, the initial algorithm is an ARE, and an associated (n-dimensional) domain D. We assume that the domain is a convex hull specified in terms of its bounding hyperplanes. Moreover, the ARE implicitly defines where the inputs (outputs) are required (computed), usually in the form of boundary conditions. Let the hyperplanes at which I/O is required be given by $\pi_1 p = \theta_1, \pi_2 p = \theta_2, \ldots \pi_m p = \theta_m$. Let $t(p) \equiv \lambda^T p + \alpha$ be a valid timing function for the ARE. As has been shown by a number of researchers (see [13], for example), any vector ξ may serve as the direction of projection for the allocation function, if it is not parallel to λ^T. We have the following theorem describing the conditions governing the behavior of the array.

Theorem 3.1 The computation of the bounding plane $\pi^T == \theta$ is performed on a set of boundary processors iff $\xi^T \cdot \pi = 0$

Due to space constraints we simply provide an outline of the proof. It is clear that since $\xi^T \cdot \pi = 0$, all the computations performed on the $\pi^T p = \theta$ plane are performed on a hyperplane of dimension $n-2$ in the processor domain. Moreover, since convex hulls are closed under affine transformations, the image of $\pi^T p = \theta$ in the space-time domain must be a *bounding* hyperplane, and hence a boundary processor. Conversely, given an $n-1$ dimensional processor array, the set of processors on the boundary are defined by an $n-2$ dimensional hyperplane. The computations performed by these processors over time are described by an $n-1$ dimensional hyperplane, which must be a *bounding* hyperplane of the array's execution trace (i.e., its space-time behavior). Moreover, since the transformations induced by the timing and allocation functions must admit an inverse (otherwise there will be a conflict), and convex hulls are closed under affine transformations, this hyperplane in the space-time domain must be the image of a *bounding* hyperplane of the initial ARE.

Hence, if all I/O is to be performed at boundary processors, then $\xi^T \cdot \pi_i = 0$ for $i = 1 \ldots m$, which is a constraint that the allocation function must satisfy. It is clear that any arbitrary domain

(particularly infinite domains.*) will not satisfy this constraint. In such a case, for each boundary $\pi_i^T p = \theta_i$ that does not satisfy $\xi^T \cdot \pi_i = 0$, we determine the vector π_i' that is perpendicular to ξ *and is in the plane defined by* ξ *and* π_i.† We *augment* the domain of the ARE so that the new boundary is given by $\pi_i'^T p = \theta_i'$, where θ_i' is the smallest integer such that $\forall x \in D$, $\pi_i'^T x - \theta_i' \geq 0$. It is easy to show that $\pi_i'^T p = \theta_i'$ is a simple rotation of $\pi_i^T p = \theta_i$, and hence any point on $\pi_i^T p = \theta_i$ may be mapped to $\pi_i'^T p = \theta_i'$ by an affine transformation. This affine transformation is the new affine dependency of the augmented domain. We are thus able to augment the original recurrence to obtain one whose I/O will occur at the boundaries of the processor array.

4 Loading and Unloading the Sort Array

Let us now return to the sort example. The derivation described in Section 2 does not include a description of how the input (output) is to be fed in (extracted from) the array. Moreover, there is no direct way to determine when the processor is to perform the specialized computation corresponding to the BC boundary. Because A'B', the image of the input boundary is parallel to the time axis and passes through $x = 1$ (the first processor), the inputs are *fortuitously* consumed at a single boundary processor. However, the output boundary BC is mapped to B'C', which means that the final results are computed in different processors at different times. We also see that the first output value is computed at point C which is mapped to the point $C' = [n, 2n-1]$ in the x-t domain.

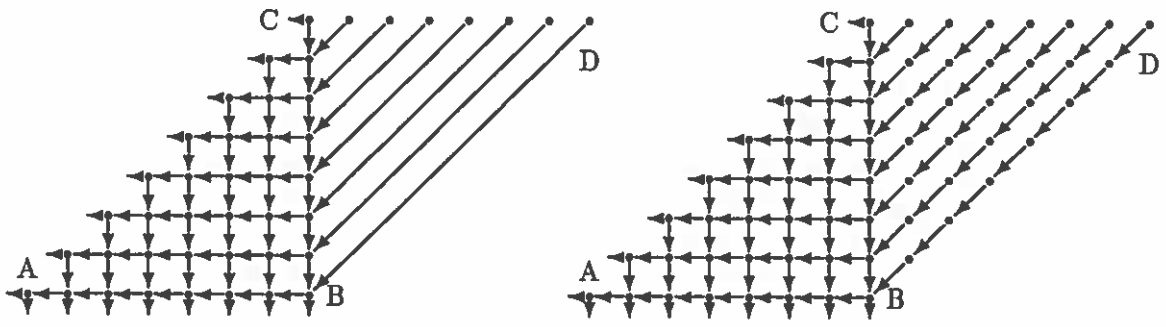
Because of our requirement that all output should occur at this processor, we must consider the set of points $\mathcal{S} = \{[n,t] \mid t = 2n \ldots 3n-1\}$ which define a straight line parallel to the time axis and passing through the n-th processor. These are the points at which the n-th processor *must* (by our thesis) output the results of the entire array. Thus, at $t = 2n + k$, the n-th processor must output the result of the $(n-k-1)$-th processor. We now determine the *inverse image* of $\mathcal{S}$, i.e., the points

$$\mathcal{R} = \Lambda^{-1}(\mathcal{S} - \alpha) = \{[n+k, n] \mid k = 0 \ldots n-1\}$$

This corresponds to the line of points, CD, as shown in Figure 4a. The new recurrence (Equa-

*In such domains, the only permissible value of ξ that yields a *finite* processor array is parallel to the ray of the domain.

†For most three dimensional, and all two dimensional domains, this vector is unique, up to scalar multiple. We choose its sign such that it points out from the domain.



a: Augmented Domain b: Pipelined Domain

Figure 4: Augmented and Pipelined Domains for the Bubble Sort Algorithm

tion 2) for the algorithm now involves the computation of $f(n+i-1,n)$ for $i=1\ldots n$ as follows.

$$f(i,j) = f_2(n, 2n-i) \tag{2}$$

where f_2 (and f_1) are as defined in Equation 1.

Analyzing the above recurrence presents a number of interesting problems. First of all, the problem domain is not a *convex hull*, and most of the earlier synthesis techniques are applicable only to such domains. This can be overcome by forming a convex closure of all the points, i.e., the parallelogram ABDC, and introducing still more points in the domain.* This is the complete domain that we shall analyze. Secondly, we see that the new dependencies are not uniform, since the point $[i,j]$ depends on $[n, 2n-i]$. The new dependency that has beeen introduced is an *affine dependency* (not to be confused with the mapping from the problem to the target domain, which is also an affine mapping). We have elsewhere studied the synthesis of systolic arrays from affine recurrences [15,17], and have shown that a systolic array may be synthesized if the dependencies can be *localized* by a technique called *explicit pipelining*. When the affine dependency of the sort specification is pipelined we get a new *uniform* dependency $[-1,-1]$ as shown in Figure 4b. The recurrence for the pipelined problem definition is as follows.

*It might that we are making the domain unnecessarily large, but we shall see that these additional points are exactly those required for the next step.

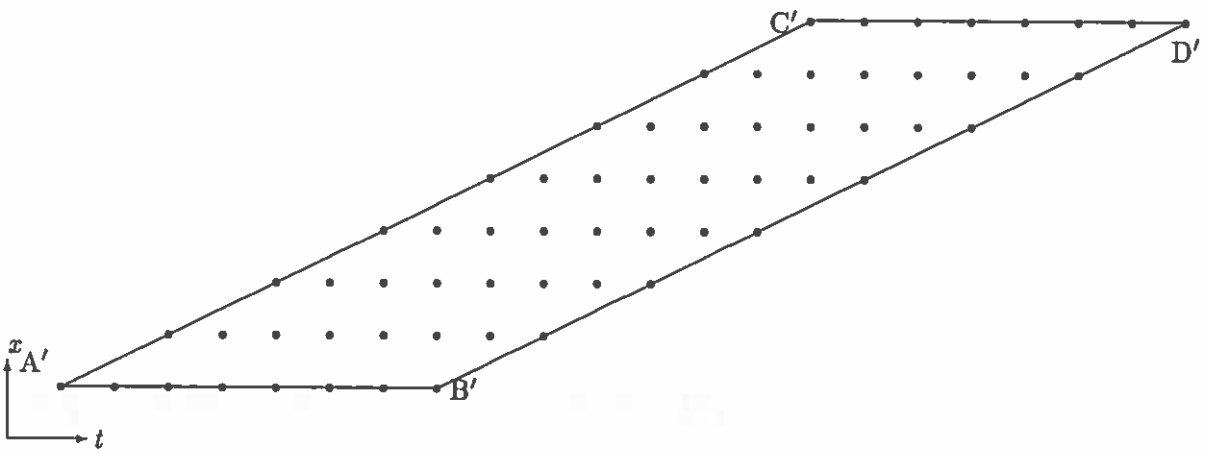


Figure 5: Target Domain for the Pipelined Algorithm

$$f(i,j) = \begin{cases} f(i-1,j-1) & \text{if } i > j \\ f_2(i,j) & \text{if } i = j \\ [f_1(i,j), f_2(i,j)] & \text{otherwise} \end{cases} \tag{3}$$

where as before f_1 and f_2 are defined as in Equation 1.

The third interesting problem with the (now uniform) sort recurrence is that the domain of computation now consists of two *subdomains*, each one a convex hull having *different* dependency structure. In addition, the computations performed in the two subdomains are completely different. In the region ABC, it consists of the old min-max function, while in the region BDC, the computation consists of simply the identity function since $f(i,j)$ is equal to $f(i-1,j-1)$. Such composite recurrences are still under investigation by this author and others [4] and a general technique for determining appropriate timing and allocation functions for them has not yet been determined. However, for the sort example, we see that the old timing and allocation functions $t(i,j) = i+j-1$ and $a(i,j) = j$ are still valid, and this yields the processor-time behavior as shown in Figure 5.

However, merely determining the timing and allocation functions is not enough. We see from the processor-time image of the computation that each processor operates in three distinct *phases*. As long as the time value does not "cross" the B′C′ line, the processor must compute f_1 and f_2 (the third line of Equation 3); at the (subdomain) boundary B′C′ it computes the second line of Equation 3 and afterwards it must compute the first line. Of these three phases, only the first and the third are significant, since the second one is merely a transition between the other two and

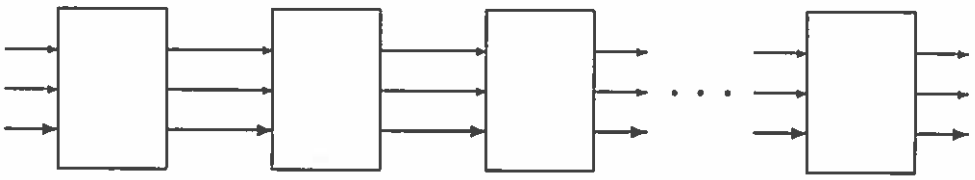


Figure 6: Derived Systolic Sorter with two control Signals

lasts for exactly one time instant. Secondly, as we have mentioned earlier, the processor must be somehow "informed" when to perform the computations on each of the domain boundaries A'C' and B'D'. We have shown elsewhere [16] that for the case when we have a single *homogeneous* domain of computation this can be achieved by means of control signals, provided the boundaries where the transition occurs is a plane (a straight line in two dimensions). In the full paper we prove that these results can be extended to the case of a composite domain provided each subdomain is also a convex hull. This results in the architecture of Figure 6.

5 Conclusions

We have addressed the problem of loading and unloading data into/from a systolic array. It is important that the problem has been addressed in the context of synthesizing such arrays from algorithmic specifications. As a result, all the architectures that are derived will satisfy an important performance criterion, namely that all I/O will occur only at boundary processors. We have presented conditions that must be satisfied by the transformations that are used to derive the architectures. These conditions are simple linear properties of the boundary equations with respect to the mapping (allocation) function. However, they may not always hold, and in such cases we have given a systematic procedure for *augmenting* the problem specification. An architecture derived from this augmented domain will have all I/O at boundary processors. Hence, when selecting a target implementation using optimality criteria, these arrays will reflect the increased cost of "initializing" the processors. We are currently in the process of incorporating these results in a design automation system.

References

[1] Engstrom, B.R. and Cappello, P.R. *The SDEF Systolic Programming System.* Tech. Rep. TRCS87-15, Computer Science Department, University of California, Santa Barbara, 1987. To appear in Journal of Parallel and Distributed Computation.

[2] Fortes, J.A.B. *Algorithm Transformations for Parallel Processing and VLSI Architecture Design.* PhD thesis, University of Southern California, Los Angeles, Ca, 1983.

[3] Fortes, J.A.B. and Moldovan, D. *Data Broadcasting in Linearly Scheduled Array Processors.* In *Proceedings, 11th Annual Symposium on Computer Architecture*, 1984, pp. 224–231.

[4] Ipsen, I.C.F. Personal Communication.

[5] Jagadish, H.V., Rao, S., and Kailath, T. *Array Architectures for Iterative Algorithms. Proceedings of the IEEE 75*, 9 (September 1987), 1304–1321.

[6] Knuth, D.E. *The Art of Computer Programming.* Addison Wesley, 1973, pp. 220–246. Section 5.3.4.

[7] Kung, H.T. *Let's Design Algorithms for VLSI.* In *Proc. Caltech Conference on VLSI*, January 1979.

[8] Kung, H.T. *Why Systolic Architectures. Computer 15*, 1 (January 1982), 37–46.

[9] Kung, H.T. and Leiserson, C.E. *Algorithms for VLSI Processor Arrays.* Addison-Wesley, Reading, Ma, 1980, chapter 8.3, pp. 271–292.

[10] Kung, S.Y. *On Supercomputing with Systolic/Wavefront Array Processors. Proceedings of the IEEE*, (1984), 867–884.

[11] Kung, S.Y., Arun, K.S., Gal-Ezer, R.J., and Bhaskar Rao, D.V. *Wavefront Array Processor: Language, Architecture and Applications. IEEE Transactions on Computers C-31*, (1982), 1054–1066.

[12] Moldovan, D.I. *On the Design of Algorithms for VLSI Systolic Arrays. Proceedings of the IEEE 71*, 1 (January 1983), 113–120.

[13] Quinton, P. *The Systematic Design of Systolic Arrays.* Tech. Rep. 216, Institut National de Recherche en Informatique et en Automatique INRIA, July 1983.

[14] Rajopadhye, S.V. *Automatic Derivation of Systolic Arrays for LU-Decomposition.* Tech. Rep. CIS-TR-87-12, University of Oregon, Computer and Information Science Department, Sep. 1987. Submitted for publication.

[15] Rajopadhye, S.V. *Synthesis, Otimization and Verification of Systolic Architectures.* PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

[16] Rajopadhye, S.V. and Fujimoto, R.M. *Systolic Array Synthesis by Static Analysis of Program Dependencies.* In *Proceedings, Parallel Architectures and Languages, Europe*, Springer Verlag LNCS No 258, Eindhoven, the Netherlands, June 1987.

[17] Rajopadhye, S.V., Purushothaman, S., and Fujimoto, R.M. *On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies.* In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag, LNCS No 241, New Delhi, India, December 1986.