SPADE: Symbolic Parallel Algorithm Development Environment

Dr. J. Greg Nash, Centar (gregnash@att.net)

ABSTRACT

This paper addresses the problem of rapidly generating optimal parallel circuit implementations from high level, formal descriptions of affinely indexed algorithms. A specialized software tool, SPADE, is described that will take a user's high level code description of his algorithms and automatically generate an abstract latency-optimal, locally connected parallel array of elemental processing elements. A design example, LU decomposition, is used to illustrate the tool's capabilities. This tool is expected to most useful in the context of building FPGA based implementations.

1. INTRODUCTION

Meeting latency and throughput requirements is a critical concern in many embedded signal and image processing applications. Consequently, a vast literature has appeared over the last two decades describing fine-grained parallel algorithms, well suited to meeting these requirements when instantiated in regular, array-based architectures that involve pipelined movement of data. Such algorithms, typically labeled generically as "systolic"[3], have been shown to be suitable for a very large range of structured problems (i.e., linear algebra, graph theory, computational geometry, number-theoretic algorithms, string matching, sorting/searching, dynamic programming, discreet mathematics).

Usage of this systolic architecture class has not been widespread in the past, in part because programmable hardware that supported this computing paradigm was not cost-effective to build. However, suitable hardware has begun to appear. Complex FPGAs now provide an adequate level of speed, density and programmability in the form of reconfigurable computers, boards, and chips with embedded computational support [12], [17], [20]. Such hardware could allow rapid implementation and change of systolic algorithms leading to inexpensive "programmable" systolic array hardware. Furthermore, the architectural characteristics of much FPGA hardware matches that required by systolic processing, because this technology is constructed from tiling identical memory and logic blocks along with supporting mesh interconnection networks.

The primary reason for the limited role of systolic processing is that to date not a single commercial software tool has appeared that allows rapid generation and exploration of systolic algorithms themselves. Without such a tool, systolic arrays are very difficult to design because

- Parallel algorithm/architecture design is an intricately complex process requiring extensive knowledge of algorithms, architectures, and hardware, and few people possess these
- Past work on finding parallel algorithm implementations are largely "point" designs that offer little insight into the numerous tradeoffs required as part of any embedded system design
- There are no systematic design methodologies with adequate generality and ease of use

The tool described here is being developed to allow a designer to easily and rapidly explore the design space of various systolic algorithm implementations so that system tradeoffs can be efficiently analyzed. The intention is to allow a user to specify his algorithm with traditional high-level code, set some architectural constraints and then view the results in a meaningful graphical format. A simulator is available that has an embedded computational model to make the transition to systolic FPGA hardware more direct. Some preliminary results described here show new tool generated optimal systolic algorithm mappings for LU decomposition, which was chosen because it has been commonly used in the past to illustrate mapping methodologies.

The intention of this paper is to provide an overview of the SPADE tool and environment, with a specific emphasis on how a circuit designer could use it to rapidly explore tradeoffs associated with a moderately complex parallel algorithm implementation.

2. RELATED WORK

Over the past 20 years much research has been directed at finding systematic methodologies for finding optimal parallel implementations of recursive or iterative algorithms. The practical motivation has always been that such systematic techniques could be reduced to a software tool, which would enable a much larger, non-experienced user community to rapidly and easily develop special purpose, fine-grained parallel computing hardware. Three basic approaches to designing relevant architectures have been proposed, those based on data dependencies [4], index transformations [9], [13], [14], and the parameter method [7].

Index transformations

This design category makes use of mathematical transformations which, when applied to systems of "uniform" or "regular" recurrence equations or their equivalent, result in parallel algorithms that represent "mappings" to an architectural model consisting of large arrays of simple, locally connected abstract processing elements (PEs). More specifically, these techniques calculate matrices that transform the index set describing the original algorithm to an index set containing at least one time dimension with the remaining indices used for spatial coordinates, i.e., a "space-time" mapping.

A classic example of such uniform algorithms is the multiplication of two matrices A and B to produce the result C:

$$\begin{aligned} &for\, 1 \le i, j, k \le N \\ &a[i, j+1, k] = a[i, j, k] \\ &b[i+1, j, k] = b[i, j, k] \\ &c[i, j, k] = c[i, j, k-1] + a[i, j, k] * b[i, j, k] \end{aligned} \quad (1)$$

where each variable element (e.g., *a[i,j,k]*) takes on a unique value (single assignment property) for each affinely referenced index vector *I*, ($I=(i,j,k)^T$). All algorithm variables are matrices or vectors. The most important characteristic of these algorithms is that all dependencies are uniform for all values of the index space *I*. For example *c[I]* above depends upon *c[I-D]*, where *D=[0,0,-1]* , for all values of *I*. Because dependence vectors like *D* will always contain only small integer values, data is inherently "localized" or pipelined in the systolic array implementation. That is, calculations associated with indices in space-time only make use of variable values obtained from the same or adjacent points in that index space.

Many mapping tools have been designed for such sets of uniform recurrence equations and the related problem of serially coded loops nests with uniform dependencies [10]. A recent effort has resulted in a tool (PICO-N), which generates synthesizable VHDL code describing embedded systolic arrays from functions expressed as loop nests in C [16].

The disadvantage of tool methodologies based on uniform dependencies is that many important algorithms (even matrix multiplication above) are not naturally expressed in this form and the process of putting them in this form can involve substantial effort because there is no systematic way of doing so.

Our tool approach generalizes the acceptable input algorithm forms to systems of "non-uniform" affine recurrence equations that closely resemble the form in which an algorithm is naturally expressed. For example in the case of matrix-matrix multiplication, it would be much more desirable to accept inputs directly in the familiar form

$$c[i,j] = \sum_{k=1}^{N} a[i,k] * b[k,j] \qquad (2)$$

rather than the cumbersome regular algorithm in (1). As can be seen in (2) there is no constant vector D that relates the dependencies between *C* and *A* and *B*. Note also that (2) is very general and unlike (1) doesn't impose detailed relationships between variables that inherently restrict subsequent choice of architectures and possibly lead to sub optimal array implementations.

A design tool, Alpha[19], based on the use of program transformations, has been proposed that is potentially capable of working from an algorithm description not restricted to uniform recurrence equations. The design process proceeds by applying a series of provably correct transformations on an algorithm coded in the strongly typed Alpha language, such that the final program form is equivalent to a uniform recurrence algorithm and can be mapped directly to an array form suitable for hardware. However, the user must be intimately involved in the choice and order of these transformations.

Data Dependencies

Rather than focus on the indices used by variables, the data dependency approach is based on creation of a "dependence graph" that shows the flow of dependencies between variables. An early tool, VACs [5], creates a visual "Space-Time" dependence graph and gives the user enough flexibility to choose how nodes in this graph are allocated to virtual processors and how to schedule the nodes. The major limitation is that the graph, and hence the original algorithm , must possess a shift invariant regularity that restricts its utility in the same way that described for uniform algorithms.

A more elaborate design methodology, the Multi-mesh Graph (MGG) method [11] does not have the regularity restrictions on the input algorithm. Although, the algorithm domain is restricted to a well-defined class of "matrix algorithms", it does accommodate non-uniform indexing functions. Two tool efforts based on this methodology, MAMACG [6] and ALIAS [8], have provided a partial demonstration of the MMG methodology. These tools generate a visually complete algorithm dependence graph to guide algorithm mapping, but it is still left for the user to decide what transformations to apply and their order. Furthermore, all variables must be heuristically embedded in this dependence graph space and there are no design guides to achieving optimality.

SPADE

The difficulty in working with non-uniform equations like (2) is that dependencies between variables are no longer local and thus the path to a systolic implementation is more difficult. There are systematic methodologies that can generate mappings with localized dependencies [15]. The localization is not uniform because the local dependencies and variables do not exist at all points in the index space. For this case there are no general techniques that will always lead to the generation of systolic solutions. Alternatively, it is possible to extend dependencies and variables to all points in the index space so that uniform recurrence equation mapping methodologies can be applied. But this leads to solutions that are sub-optimal because extra variables and dependencies are introduced. Also, user intervention is still required.

In order to solve the general problem of non-uniform affine recurrence equations when desirable architectural constraints are imposed, it is necessary to use non-linear integer programming methods [2]. Two tool efforts that have been proposed for this class of algorithms both solve the integer-programming problem by structuring them as customized searches [1],[18]. The focus in Cathedral IV [18] is to maximize processor utilization for problems with a given throughput and input-output scheme using as a metric a sum of dependence lengths. Alternatively, DESCARTES [1] uses algorithm latency as an objective function metric. One important difference between the two tools is that the DESCARTES search is structured to find optimal designs, whereas the Cathedral search process is more limited and optimal designs aren't guaranteed. Optimal parallel solutions are critically needed for embedded applications where system size, weight, power, and volume constraints drive costs extremely high for non-optimal solutions. DESCARTES places strong reliance on the use of architectural constraints to reduce the space of possible systolic solutions, making the search strategy feasible.

SPADE uses a search methodology that is based on that used in DESCARTES, but involves a different formalism and is organized to provide better coverage of the architectural solution space for cases where solutions are less architecturally constrained. It also incorporates a parser to handle high level nested loop inputs, does additional analysis of potential solutions to give the designer more control over design tradeoffs, includes a simulator that uses an embedded model of

computation and is instrumented to provide useful architectural statistics.

3. TOOL DESCRIPTION

3.1 Introduction

Past difficulty in obtaining optimal solutions for parallel algorithm implementations has been due to the many complex, interrelated steps that are involved and the lack of any systematic approaches to dealing with them:

1. **Algorithm representation:** How a user describes his problem to the software tool involves various tradeoffs. For example, as mentioned in Section 2, the requirement that dependences be uniform makes writing code harder but simplifies the mapping process.
2. **Scheduling:** Finding an execution sequence for the pieces of an overall computation (e.g., making sure that all variables necessary to do the computation are available at the right PE at the correct time).
3. **Reindexing:** Altering the indexing relationships between different variables to best expose regularity.
4. **Localization or regularization:** The process of decomposing global dependencies, like "broadcast" and "fan-in" operations, into sequences of local data movement, so that all data movement between PEs can be performed by a nearest neighbor connection network.
5. **Allocation:** The process of allotment of computations to virtual processors in spatial coordinates

The steps above are not independent because a choice made at any step can affect those at other steps. For example, allocation and scheduling steps would be in conflict if two computations with the same scheduled execution time were allocated to the same processor. The "reindexing step" has been particularly difficult because it involves making choices in the indexing relationships between variables. Historically users have had to make such choices based on previous experience and knowledge of an algorithm's peculiarities.

There is no prescribed order to the steps above and different mapping methodologies sequence the steps differently. For example use of uniform affine recurrence equations requires that the localization process occur first because it is inherent in the algorithm description.

SPADE performs space-time mapping first. This is a combined search process that examines scheduling, reindexing and allocation in that order. After all minimum latency solutions are found, an attempt is made to localize these solutions. If this isn't possible, then the next best minimum latency space-time mapping solutions are chosen and the process repeated until at least one solution is found.

3.2 Space-Time Mapping

SPADE takes as input a coded form of non-uniform recurrence equations. This section provides a precise statement for this and summarizes the concept of space-time mapping.

A formal description of non-uniform recurrence equations has been provided by Roychowdhury [15], where they are referred to as "linear indexed weak single assignment codes". An equivalent description is given Baltus [1], where they are termed conditional affine recurrence equations (in this context "conditional" is intended to imply that each equation can have its own unique index space). A system of non-uniform affine recurrence equations is

$$\begin{aligned} w_1(A_1(I)) &= g(\dots w_i(B_{i1}(I)),\dots) \qquad \textit{for all } I \textit{ in } \mathrm{I}_1 \\ &\dots \qquad\qquad\qquad\qquad\qquad\qquad (2) \\ w_n(A_n(I)) &= f(\dots w_i(B_{in}(I)),\dots) \qquad \textit{for all } I \textit{ in } \mathrm{I}_n \end{aligned}$$

where *f,g* represent the functional variable dependencies, I_j is the index range for equation *j*, w_i is one of the algorithm variables and the affine indexing functions are

$$\begin{aligned} A(I) &= AI + a \\ B(I) &= BI + b \end{aligned}.$$

Here, *A/a,* and *B/b* are integer matrices/vectors. All assignments of values to variable elements in the system of the equations must not involve a reuse of a variable.

For each algorithm variable SPADE finds an affine transformation, *T* that maps this algorithm variable's indices to space-time, e.g., for a variable *x*:

$$T(x) = T_x(A_x I + a_x) + t_x \qquad (3)$$

where T_x is a matrix and t_x is a vector. Thus, every variable gets mapped to a unique point in the space-time domain.

The transformation *T* can be thought of as consisting of two parts, one that determines the scheduling index and one that determines the spatial index. That is, writing $T(x)$ using

$$T_x = \begin{bmatrix} \Lambda_x \\ S_x \end{bmatrix}, \qquad t_x = \begin{bmatrix} \gamma_x \\ s_x \end{bmatrix} \qquad (4)$$

means that variable $x[I^T]$ would be mapped to a time index $\Lambda_x(A_x I + a_x) + \gamma_x$ (Λ_x / γ_x is a vector/scalar), and to a spatial index $S_x(A_x I + a_x) + s_x$ (S_x / s_x is a matrix/vector with a number of rows/elements equal to the dimension of the spatial array).

Each time index corresponds to potential activity (data transfer or calculation) in all PEs with that same index value. The basic execution cycle for a time index value consists of two steps: first there is a local movement of data between adjacent PEs, followed by a computation step in those PEs. The algorithm latency is the total number of these time steps needed to compute the entire result.

3.3 Solution search

From (4) it is clear that to specify the space-time mapping for *x* it is necessary to find the elements of Λ_x, S_x, s_x and the scalar γ_x. For example, given $\Lambda_x = [\lambda_{x1} \quad \lambda_{x2}]$, SPADE considers λ_{x1} and λ_{x2} as "search" variables (as distinct from previously mentioned "algorithm variables"). Following Baltus [1], the

allocation matrices like S_x are treated as a single variable, each a unimodular matrix to ensure that the space-time mapping is "dense". The small dimensionality of *S* limits the number of unique unimodular matrices that have to be considered. The fact that each algorithm variable has a different spatial mapping *S* is equivalent to it being reindexed with respect to other algorithm variables.

Finally, given a set of search variables, SPADE examines all possible combinations and chooses those that produce the minimum algorithm latency. The difficulty in doing this is that there are potentially a large number of these search variables and even though each need take only a small range of values, the search can be computationally infeasible. Baltus [1] solves this problem by introducing computational and architectural constraints that limit the space of solutions that has to be searched. For example, causality requires that a computation can't occur if its arguments are not available. From the example from (2) it can be seen that computation of *c* depends upon input *a*. Thus, it must follow that temporally

$$\Lambda_c(A_c I + a_c) + \gamma_c - \Lambda_a(B_a I + b_a) - \gamma_a \geq 0 \qquad (5)$$

With a large number of dependencies this generates a large number of such constraints.

Architectural constraints are just as important. For example, if it is desired that input of data occur from the edge of the PE array, this means that the position in space-time of an input matrix, such as *a[i,j]* in (2) , must be such that the unit time vector u_t is in the plane of the PE array (hence it's projection on the PE array is a line). This constraint requires that the normal n_a to the plane of *a* satisfy $u_t \bullet n_a = 0$. Also, to be at the edge of the PE array it must not be within the convex hull of the array. Such constraints as these considerably limit choices for Λ_a and S_a

The search strategy used in SPADE is similar to that of DESCARTES but with several modifications. For example, DESCARTES's search is depth first with a suitable ordering of search variables that maximizes its efficiency. This worked best when desired architectures were highly constrained and the solution space was small. However, when fewer constraints are involved, it was found to be more computationally efficient to introduce a degree breadth first processing, i.e., often the overhead from invoking a complex matrix computation is large enough to justify processing all matrix search variables at that search depth rather than the one search variable in a depth first approach.

3.4 Input

Input to SPADE is in the form of high-level code based on an Algol-like language that is a subset of the Maple[1] programming language. Very often it is possible to go directly from a scientific expression to equivalent code because the Maple language provides special syntax options for the commutative and associative operators (multiply, add, minimum, maximum) this tool supports. For example, the matrix-matrix algorithm (2) can be written in this language directly as

```
for i to N do
   for j to N do
      c[i,j] := add(a[i,k]*b[k,j],k=1..N)
     end do;
end do;
```

where the Maple "add" construct directly replaces the mathematical summation sign. This code as written runs correctly in both SPADE and Maple. Maple treats the loop structure in traditional way, but SPADE does not make any lexicographic interpretation of the loops; rather it uses the loop limits only to determine the index space of the inner statement body. Computational ordering is determined directly from the loop body.

It does not always happen that code written for SPADE will run correctly in Maple and vice-versa; so the user must be wary of the distinctions between recurrence equations and traditional serial code. The goal was just to make it as easy as possible to go back and forth between serial and parallel expressions of algorithms when possible.

When conditionals are used in code input to SPADE, they override the loop limits in determining the index space. And when using the simulator described in Section 3.6 it is important that this form be used because it makes explicit how the index space and operations are arranged.

Other inputs pertain to architectural constraints desired and objective function criteria used to select solutions from the search space. Architectural constraints specify (1) which variables should be constrained to align with the time axis and/or the PE array boundary and (2) whether overlap of variables in space-time is allowed. Selection criteria picks out minimum latency solutions with (1) minimum area, (2) maximum regularity and (3) minimum array bandwidth.

3.5 Output

SPADE's primary outputs are the values *T,t* for each of the algorithm variables. In addition a set of vectors are computed indicating the direction of data flow for each dependency in the algorithm. For the example (2) it can be seen that *c* needs input from *a*, so corresponding to this dependence is a uniform flow of data from *T(a)* to *T(c)* in the space-time domain. Since this flow is one dimensional, a vector v_{ca} is calculated to indicate its direction. Consequently, every point in space-time associated with *T(a)* represents a source of a data element moving in direction v_{ca} to a corresponding point in *T(c)*. The transformations *T* are such that this data element arrives just in time to be used (along with other data) in a calculation that produces an element of the result *c*.

Unfortunately, the mathematical specification of *T* provides little insight into the nature of the solution, especially from the designer's point of view. For this purpose graphical tools have been included as part of SPADE. The two primary mapping views are (1) of space-time, which shows placement of the mapped algorithm variables, and (2) of the spatial mapping only, overlaid by the projected data-flows associated with all algorithm dependencies. When the space-time views are three

[1] Maple is a commercial "scientific" programming environment (Waterloo Maple Inc).

dimensional, it is helpful to rotate objects in real-time to "see" alignment of data sources with corresponding destinations.

3.6 LU Decomposition Example

3.6.1 Algorithm Derivation

In this section LU decomposition is used as an example to show in detail how arrays can be derived directly from mathematical expressions. LU decomposition also serves as a useful benchmark in comparing previous tool and methodology efforts because many of these use it as an example. The goals of this section are to show how little user intervention is required in the design process and how easy it is for a circuit designer to easily explore a variety of array implementation tradeoffs.

Given a linear system $Ax=b$, where A is a known NxN matrix , b is a known vector, and x is an unknown vector, a common solution is to decompose A into a product of two matrices, one lower triangular in form (L) and the other upper triangular (U), so that $LUx=b$. This allows the linear system to be decomposed into two easier to solve systems,

$$\begin{aligned} Ly &= b \\ Ux &= y \end{aligned} \qquad (6)$$

where another another unknown vector y has been introduced. Each equation in (6) can then be solved by the process of "back substitution". For example, $Ly=b$ can be written in the matrix form

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

whereupon it can be seen that 3 recursive expressions in y allow the solution

$$\begin{aligned} y_1 &= b_1 / l_{11} \\ y_2 &= b_2 - l_{21} y_1 \\ y_3 &= b_3 - l_{31} y_1 - l_{32} y_2 \end{aligned}$$

or in general for L

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k \,.$$

A similar expression provides the solution x now that y is known. Thus, using this approach, all that is required to solve the linear system is to decompose A into the product LU. Such a decomposition can be done in a fashion similar to that above, which can be seen directly from the form

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Hence, it can be determined from inspection that

$$\begin{aligned} & l_{11} = a_{11};\ l_{21} = a_{21};\ l_{31} = a_{31}; \\ & u_{12} = a_{12} / l_{11};\ u_{13} = a_{13} / l_{11} \\ & and\ for\ i \geq j, j > 1, i \leq 3 \\ & l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j} \\ & and\ for\ j > i, i > 1, j \leq 3 \\ & u_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{i,k} u_{k,j}) / l_{i,i} \end{aligned} \qquad (7)$$

From the mathematical expression in (7) above it is possible to go directly to the Maple code form:

```
for i to N do
  for j to N do
    if j=1 and i>=1 and i<=N then
      l[i,j]:=a[i,j];
    elif i=1 and j>1 and j<=N then           (8)
      u[i,j]:=a[i,j]/l[i,i];
    fi;
    if i>=j and j>1 and i<=N then
      l[i,j]:=a[i,j]-add(l[i,k]*u[k,j],k=1..j-1)
    fi;
    if j>i and i>1 and j<=N then
      u[i,j]:=(a[i,j]-  \
        add(l[i,k]*u[k,j],k=1..i-1))/l[i,i]
    fi;
  od
od;
```

Here, the only semantic change has been to use the Maple "add" construct in the place of the mathematical summation sign. Also, N replaces "3" and is a measure of the "problem size". In this example it is possible to run this code directly in Maple to verify it's functionality.

This code is saved as a text file and read into SPADE as a text file at the beginning of processing by the parser. All variables are required to be indexed and of dimension less than four. There is nothing in the space-time mapping methodology in Section 3.2 that imposes this limitation; rather it was a practical choice based on the observation that there are no FPGA based 3-D virtual computers.

Thus, it can be seen that a suitable algorithm input to SPADE can be derived from first principles and can be used directly as an input. In this case there were no steps in this process involving issues related to parallel algorithms or relating to architectural implementation issues.

In contrast a uniform or regular LU decomposition algorithm form required by the index and dependency methods described in Section 2 would look like

$$
\begin{aligned}
&for\ i>=1,\ i<=N,\ j<=N,\ j>=1,\ k=1\\
&\qquad a[i,j,k]\ :=\ A[i,j];\\
&for\ i>=k+1,\ i<=N-1,\ j<=N,\ j>=k,\ k>=1,\ k<=N-1\\
&\qquad u[i+1,j,k]\ =\ u[i,j,k]\\
&for\ i=k,\ j<=N,\ j>=k,\ k>=1,\ k<=N-1\\
&\qquad u[i+1,j,k]:=\ a[i,j,k]\\
&for\ i>=k,\ i<=N,\ j<=N-1,\ j>=k+1,\ k>=1,\ k<=N-1\\
&\qquad l[i,j+1,k]\ =\ l[i,j,k]\\
&for\ i>=k+1,\ i<=N,\ j=k,\ k>=1,\ k<=N-1 \qquad\qquad (9)\\
&\qquad l[i,j+1,k]\ =\ a[i,j,k]/u[i,j,k]\\
&for\ i>=k+1,\ j>=k+1,\ i<=N,\ j<=N,\ k>=1,\ k<=N-1\\
&\qquad a[i,j,k+1]\ =\ a[i,j,k]\ -\ l[i,j,k]*u[i,j,k]\\
&for\ i=k,\ j<=N,\ j>=k,\ k>=1,\ k<=N\\
&\qquad U[i,j]\ :=\ a[i,j,k]\\
&for\ j=k+1,\ i>=k+1,\ i<=N,\ k>=1,\ k<=N-1\\
&\qquad L[i,j]\ :=\ l[i,j,k]
\end{aligned}
$$

Since there is no systematic design methodology to achieve such algorithm forms, it was necessary here to introduce a 3rd index, *k*, as part of the regularization process. In addition detailed relationships between the indices of each algorithm variable have been heuristically imposed. In effect the parallel architectural implementation is almost finished at the algorithm level, leaving no possibility to explore other architectural implementations that might otherwise be optimal.

3.6.2 LU Decomposition Design Results

A critical tool function is to provide the circuit designer with a complete range of architectural options so that tradeoffs can be adequately analyzed. This is necessary because available FPGA chips, boards and virtual computers are built with different architectural features and any system implementation based on such technologies will have its own unique set of constraints. The purpose of this section is to provide a few examples of how such considerations can result in very different array designs.

3.6.2.1 Input Constrained to Array Boundary

In this first mapping example the architectural constraints were set to require all input to take place from the systolic array boundary and look for the minimum time latency solution that has the least PE array area. (In this case variable *a* is the input.) It is a common characteristic of systolic arrays, which are physically tied to sensors to have sensor data stream into the array from one or more array edge boundaries, and also consistent with FPGA based virtual computers that contain a lot of buffer memory at the board inputs.

With the LU decomposition input from (8) the parser analysis generates 55 search variables that are uniquely associated with the scheduling operation. However, the causality constraints (5) and other high level architectural constraints reduce the number of these search variables to 19. There are also 6 unimodular matrices associated with the reindexing step that in part specifies *S*, and these matrices must be chosen from a set, typically less than 50. When all high level constraints applied, the search space of possible mappings is reduced to a range that can be explored in a reasonable length of time.

The search discovered 303 solutions that satisfied causality. These can be grouped into just 2 time latency categories proportional to 3N and 4N. However, when additive constants are considered (e.g., 4N-3) there were 8 distinct latency values for all solutions. Constraints associated with the boundary I/O and reindexing reduced the total number of feasible solutions to 14 and finally after all constraints were considered there were only six unique mappings with the minimum time latency of 3N-3. These array designs were all triangular in shape. Four of these had less desirable "diagonal" interconnection patterns between PEs. Going through the search process again with the secondary constraint set to find the most regular solutions, yielded just two designs that have the more desired "rectangular" interconnect structure. These two designs are shown in Figures 1 and 2 for the case where N=6.

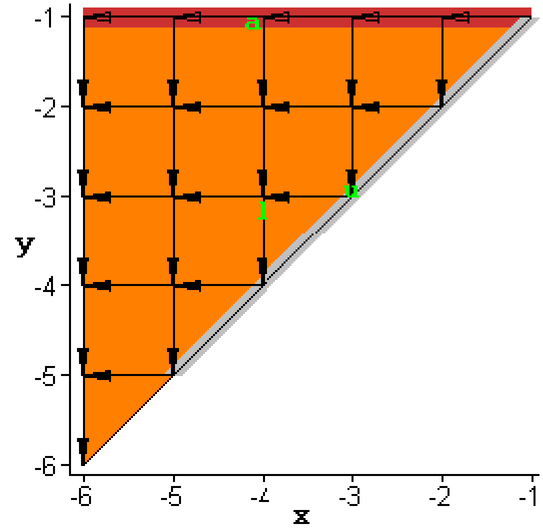

Figure 1. The first of two mappings for LU decomposition with input variable a constrained to appear on the array boundary (N=6).

Here, as with later mappings, the *x* and *y* axes represent the spatial coordinates of a 2-D mesh grid with the array of PEs embedded in this grid at intersection points. The grid above is NxN in size and the embedded systolic array is triangular with N(N+1)/2 PEs. The different shadings correspond to regions associated with the different algorithm variables *a*, *l* and *u* and are labeled as such[2].

The PE arrays in Figure 1 and 2 are uniform in terms of the interconnection pattern and there are six different flows of data associated with the various variable dependencies, each of which moves along an orthogonal path defined by the arrows in Figure 1 and 2. Thus, some PEs experience six different data streams passing through them. There are three additional dependencies in which data does not move spatially, but rather is updated and reused in the same PE. This corresponds to data

[2] The algorithm variables *a*, *l* and *u* will appear thoroughout as red, orange and gray, or in black-and-white as heavily shaded, medium shaded and lightly shaded.

"movement" along the time axis in space-time. The picture in Figures 1 and 2 show a superposition of data flow at all times. The actual time variation of data flow is more complex, with the size of uniform sub-regions of PEs growing and shrinking with time. In other words data movement typically begins at one edge of the array and proceeds outwards in a "wavefront-like" fashion.

In space-time all three algorithm variables, *a*,*u* and *l*, are represented by polygons because they are defined with two indices in the algorithm code (8). That is, the index points in space-time to which the element values *a[i,j]*, *u[i,j]* and *l[i,j]* are mapped lie on a planar surface. When the normal to these surfaces is perpendicular to the time axis, they are projected onto the 2-D mesh array as lines as shown in Figures 1 and 2. In this first mapping example such a constraint was only placed on certain search variables associated with the elements of input variable *a*. (The boundary alignments of *u* in Figure 1 and *l* in Figure 2 occurred only because this lead to the most regular array.)

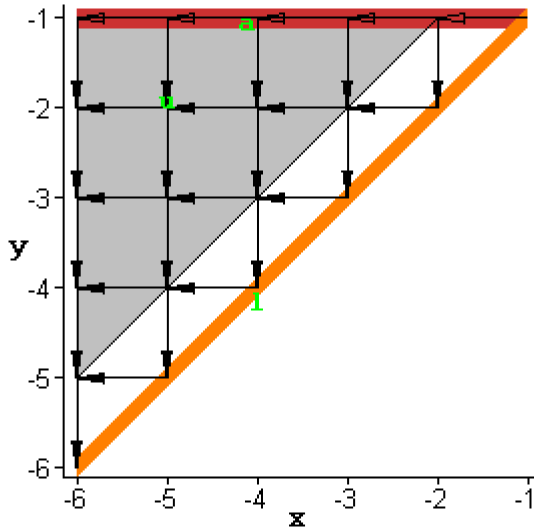

Figure 2. The second of two mappings for LU decomposition with input variable a constrained to appear on the array boundary (N=6).

SPADE also provides a space-time view of each algorithm array mapping solution, one of which is shown in Figure 3 and corresponds to the spatial-only view in Figure 1. The space-time view is shown from two different perspectives in order to help in its interpretation (in the SPADE environment this view can be easily manipulated along all axes in real-time).

Figure 3 is more complex because it shows additional variables, *IM1* and *IM2* (*IM1[i,j,k]=l[i,k]*u[k,j]*; *IM2* has the same dependence on *l* and *u*, but a different index domain and both are polytopes in the space-time view) not seen in the original algorithm (8). These two new variables are created automatically [1] in the parser and are there to keep running sums associated with the summations in (7).

This view imparts a good deal more information than the spatial view. For example it shows where and when array activity associated with the different algorithm variables takes place, it provides a visible view of 3-D data flow between algorithm variables, and it imparts a rough estimate of how efficiently PEs are used by what percent of the total space-time volume is occupied by polytopes and polygons.

In Figure 3, it can perhaps been seen more clearly that the normal to the polygons representing variables *a* and *u* in space-time is in the plane of the PE array and therefore their projection on this plane is a line.

The design significance of placing a variable entirely along an edge boundary goes beyond its proximity to the rest of the system, which is external to the array. It means that each PE associated with one of the edge points on the grid must have a memory size that's O(N), where N specifies the size of the problem. This requirement is different from PEs internal to the array for which the memory requirement is small and independent of the problem size.

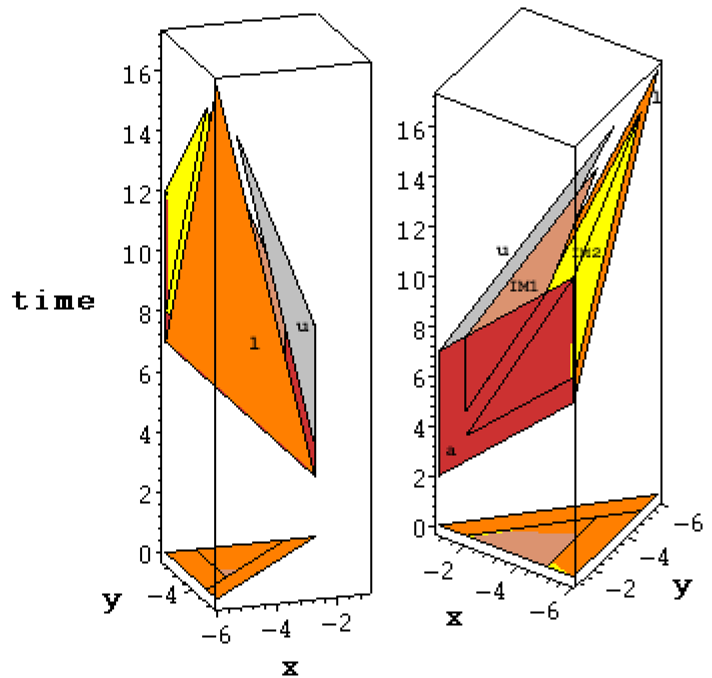

Figure 3. Position of algorithm variables *u*, *l*, and *a* in space-time for two different orientations. Projection of these surfaces along the time axis yields the triangular array of Figure 1. The outline of this projection is shown here at time=0.

Given the considerations above, the circuit designer might wonder if there is an architecture that doesn't place either *l* or *u* at the edge of the array as in Figure 1 and 2. This could be potentially motivated by the desire to have all the output data internal to the array for subsequent processing. Such a solution does not appear amongst those that fall into the minimum area category. But by having SPADE generate some of the more sub-optimal area solutions, the desired result can be found and is shown in Figure 4. Cleary the tradeoff in accepting the array shown in Figure 4 is that it is about twice the size of the previous designs. Note that the "gap" between the *l* and *u* regions is a result of the fact that the diagonal elements of u are by definition already set to one.

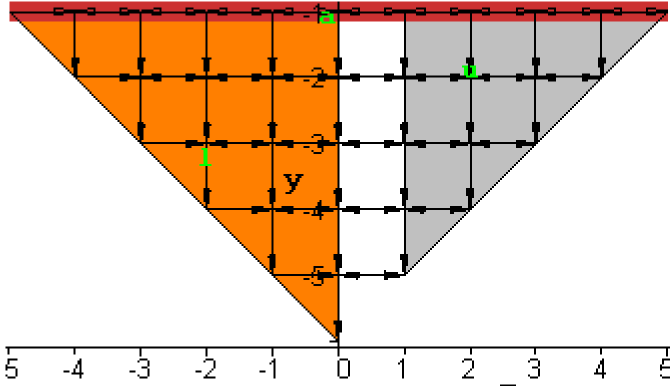

Figure 4. Array design with variables l, u not placed on array edge boundary having a time latency of 3N-3 (N=6).

3.6.2.2 Input and Output Constrained to Array Boundary

Naturally, it might be of interest as well to inquire about array designs would occur if *l*, *u* and *a* were all constrained to occur along the boundary. Only two solutions were found as shown in Figure 5. However, the latency of the algorithm with this new constraint has increased from the 3N-3 time steps associated with all designs in Section 3.6.2.1 to 4N-4 time steps. In addition the area has increased from N(N+1)/2 to (2N-1)N.

It is likely that a circuit designer would prefer working with the array design on the right in Figure 5 because here the actual PE array is square and since most FPGA hardware arrays are orthogonal in organization. (Sometimes it is necessary to view the SPADE space-time output in order to ascertain exactly which grid intersections of the 2-D view correspond to PEs.)

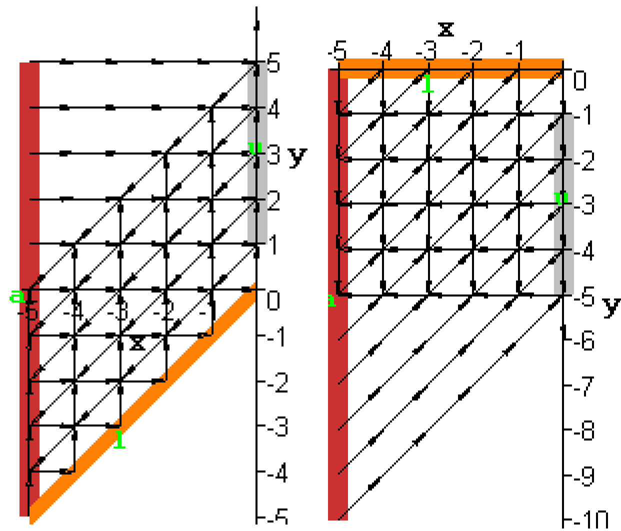

Figure 5. Two optimal arrays resulting from constraints that all input and output occur on array boundaries (N=6).

3.6.2.3 Single Divider Implementation

Although the array designs in Figure 1 and 2 look equivalent there is a significant difference between the two in terms of hardware usage. From (8) it can be seen that the line

```
if j>i and i>1 and i<=N and j<=N then
  u[i,j]:=(a[i,j]-  \                               (10)
      add(l[i,k]*u[k,j],k=1..i-1))/l[i,i]
fi;
```

implies that at each point *i>1,j>1,* a divider is necessary to compute this statement. Consequently, in Figure 2 a triangular array of dividers corresponds to the internal shaded region labeled *u*, whereas in Figure 1 only a linear array associated with the linear projection of *u* is required.

Since it is known that systolic arrays that do LU decomposition are possible that use only one divider it is useful to ask how SPADE would achieve such a result. This is best done by introduction of a new variable. This is necessary because it is clear that if a variable positioned in space-time is to project onto the spatial plane at a single point (corresponding to the divider), this variable can't be represented by a polygon, but must be represented by a line. Hence it must be specified by a single index. Also, it is clear in (10) that although there are $O(N^2)$ divides associated with this statement, only N values of *l[i,i]* used and this could be specified by a single index. Therefore it follows that, the number of divisions can be reduced by changing (10) to read

```
if j>i and i>1 and i<=N and j<=N then
  u[i,j]:=(a[i,j]-  \
      add(l[i,k]*u[k,j],k=1..i-1))*l_inv[i]
fi;
if i>=1 and i<=N then l_inv[i]:=1/l[i,i] fi;
```

and replacing instances of division by *l* to multiplication by *l_inv* elsewhere in (8). If this change is made along with a boundary constraint on *l_inv*, we get the single solution shown in Figure 6 with an area of N^2 (minimum area secondary objective function) and time latency of 3N-3. The corresponding space-time view is shown in Figure 7.

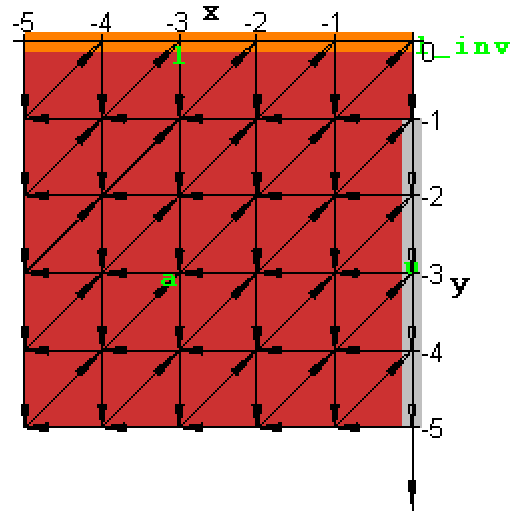

Figure 6. Optimal array implementation for single divider (upper right hand corner) (N=6).

The tradeoff above is that the single divider constraint increases the array size by about a factor of two compared to the array design in Figure 1. In addition, the placement of variables *l*, *u* and *a* in this design might or might not be considered favorable.

Also there's data movement in nine vs the six directions for the array in Figure 1. But, while the creation of a new variable often increases the overhead in terms of data movement and computational requirements, this doesn't happen here because the variable elements *l_inv[i]* remain in the same PE (no data movement) and the number of total divisions has not increased compared to previous examples.

Once an array design is selected, the actual transformation matrices are saved and available as inputs to other components of the environment, in particular the simulator (Section 3.7) and the VHDL translator (Section 3.8). For the particular design shown in Figures 6 and 7 these are summarized in Table 1 and the dependency information (data flow vectors) is provided in Table 2.

3.7 SIMULATOR

There is also a simulator that can be run to mimic operation of the systolic array. It takes as inputs the various mathematical quantities that specify the algorithm mapping and then simulates activity in each PE during the systolic computation. Each simulator time step consists of two phases. First there is data transfer between PEs; then there is (potentially) a computation within each PE.

This is a very abstract model of a real systolic array; that is, it presumes there is a data path available for all data flow, that there is a physical PE for each virtual PE, that there is no PE control "overhead", and that all computation can be done in one-half time step.

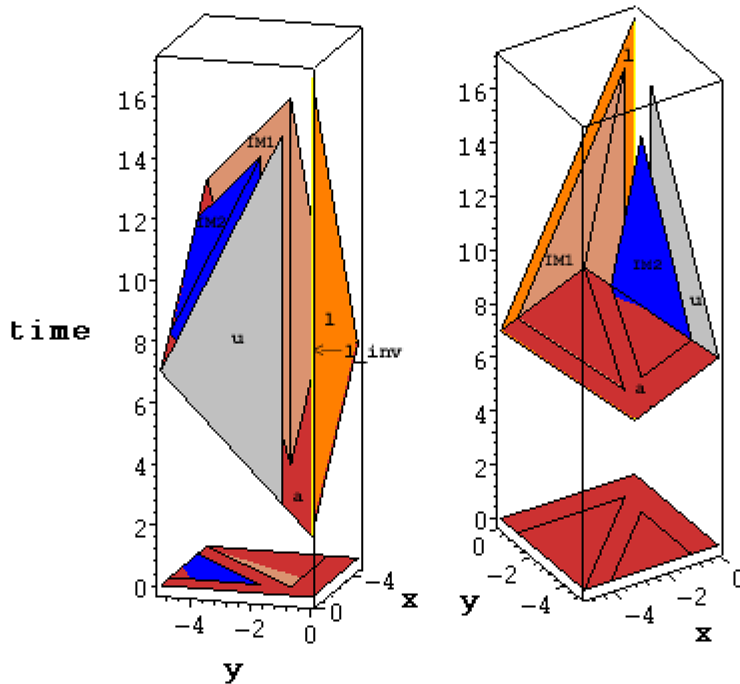

Figure 7. Space-time view of single divider array design corresponding to spatial array shown in Figure 6. Variable l_inv is the lightly shaded (yellow) vertical line at x=0, y=0.

However, the simulator does represent a useful model of computation. In particular it is initialized in somewhat the same manner a real systolic array would be and the simulation mimics the actual flow of data and abstract computation activity in each

For example, sets of registers are defined that are associated with specific data dependencies, such that when a data item is transferred, its direction of propagation is included along with the data. This information is used to control its movement during subsequent time steps. Also, each PE is initialized with (or without) information about what computations it performs. In a real implementation PE control could possibly localized with each PE having its own unique finite state machines. Thus, with instrumented output from the simulator, e.g., PE efficiency, the transition to real hardware could be facilitated.

4. VHDL TRANSLATOR

Although it has not yet been completely implemented we have built a prototype translator that generates behavioral VHDL code and is perhaps suitable for generating input to a commercial behavioral compiler which could generate FPGA output designs in a similar way to those that generate an ASIC output.

The translator is based on the use of a "template" that describes in a generic way the desired VHDL code or architecture. Using this template, a Maple a "compiler" was written that takes architectural specifications from a data file (an output from our tool) and instantiates this in another form of a Maple program. When this Maple program is run, the actual target code (VHDL) is generated.

Table 1. Transformations for array design of Figures 6 and 7.

variable	T	t
a	$\begin{bmatrix} 1 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$
u	$\begin{bmatrix} 2 & 1 \\ 0 & 0 \\ -1 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
l	$\begin{bmatrix} 1 & 2 \\ -1 & 1 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
IM1	$\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
IM2	$\begin{bmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$
l_inv	$\begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$

The advantage of this two-stage translator process is that it separates the template step, where the target architecture is

described, with the process of conversion of this description to target code. Consequently, the template is very readable and easily modified to reflect changes in the architecture. Doing the entire translation in one step would result in a complex and clumsy arrangement, difficult to understand or modify.

Table 2. List of direction vectors (*[time,x,y]*) of data flow for the various dependencies corresponding to the transformation matrices in Table 1.

dependency	v
$l \rightarrow a$	$[1 \; 1 \; 1]$
$l \rightarrow IM1$	$[1 \; 1 \; 1]$
$u \rightarrow a$	$[1 \; 1 \; 1]$
$u \rightarrow l_inv$	$[1 \; 0 \; -1]$
$u \rightarrow IM2$	$[1 \; 1 \; 1]$
$l_inv \rightarrow l$	$[0 \; 0 \; 0]$
$IM1 \rightarrow l$	$[1 \; 0 \; -1]$
$IM1 \rightarrow u$	$[1 \; -1 \; 0]$
$IM2 \rightarrow l$	$[1 \; 0 \; -1]$
$IM2 -> u$	$[1 \; -1 \; 0]$

5. SUMMARY

Although FPGAs conceptually provide an excellent implementation strategy for systolic arrays, a designer faces a large number of design options, given the variety of reconfigurable computers, boards and chips that are becoming available. For a given algorithm different systolic array mappings will present different tradeoffs from which he can make optimal choices. Further design complexity is introduced when building programmable systolic arrays that support several different algorithms. In this case a designer must consider architectural tradeoffs among systolic array mapping possibilities for each of the different algorithms he wants to support. Having available only a few systolic "point" designs will lead to sub-optimal implementations. Consequently, the utility of this tool is that it can facilitate identification of such tradeoffs rapidly and easily.

It should be noted that the exhaustive search procedures used in SPADE will not always guarantee mapping solutions. The reason for this is that dependencies expressed the algorithm code might preclude a mapping; in such cases alternative codings must be explored. The primary goal of SPADE is to move the level of abstraction at which the designer must work away from the realm of detailed architectural and timing issues to the more familiar world of high-level code. An example of how this might be accomplished was shown in the case of the single divider design, where the addition of one more line of code led to the desired result.

6. ACKNOWLEDGMENTS

This work was supported in part by DARPA Contracts DAAH01-96-C-R135 and DAAH01-97-C-R107.

7. REFERENCES

[1] Baltus, Donald and Allen, Jonathon, "Efficient Exploration of Nonuniform Space-Time Transformations for Optimal systolic Array Synthesis," Proc. Application specific Array Processors, 1993, pp.428-441.

[2] Feautrier, Paul, "Fine Grain Scheduling under Resource Constraints," 7th Workshop on Language and Compilers for Parallel Computers, Aug. 1994.

[3] Kung, H.T., "Why Systolic Architectures?", IEEE Computer Magazine, vol 15., pp. 37-45, Jan. 1982.

[4] Kung, S.Y., "VLSI Array Processing", Prentice Hall, 1988.

[5] Kung, S.Y., and Jean, S.N.,"A VLSI Array Compiler System (VACS) for Array Design",Proc . IEEE Workshop on VLSI Signal Processing, pp.495-508,1988.

[6] Le, Dinh, et.al, "MAMACG: A Tool for Automatic Mapping Matrix Algorithms Onto Mesh Array Computational Graphs", Proc. 1992 Application Specific Array Processors, IEEE Computer Society Press, p. 511-525.

[7] Li, G.I., and Wah, B.W., "The Design of Optimal Systolic Arrays," IEEE Trans. on Computers, Vol. C-34, pp. 66-77, Jan. 1985.

[8] Liu, James and Ercegovac, M.D., "ALIAS Environment: A Design Tool for Application Specific Arrays", In Fifth IEEE Symposium on Parallel and Distributed Processing, Dec. 1993.

[9] Moldevan, D. I., "Parallel Processing", Morgen Kaufmann, 1993.

[10] Moldevan, D.I., "ADVIS: A Software Package for the Design of Systolic Arrays," IEEE Trans. on Computer-Aided Design, pp. 33-40, 1987.

[11] Moreno, J.H. and Lang, T., "Matrix Computation on Systolic-type Meshes: An Introduction to the Multi-mesh Graph Method", IEEE Computer, 23(4):32-51, April 1990.

[12] OptiMagic, Inc; http://www.optimagic.com/boards.html

[13] Parhi, Keshab K, "VLSI Digital Signal Processing Systems", John Wiley, 1999, Chapter 7.

[14] Quinton, P. and Robert, Y., "Systolic Algorithms and Architectures", Prentice Hall 1991.

[15] Roychowdhury, V.P., S.K. Rao, L. Thiele, and T. Kailath, "On the Localization of Algorithms for VLSI Processor Arrays," VLSI Signal Processing III, IEEE Press, 1988, pp. 459-470.

[16] Schreiber, R., et. al., "High-Level Synthesis of Nonprogrammable Hardware Accelerators", Proc. IEEE Int. Conf. Application Specific Systems, Architectures, and

Processors, IEEE Computer Society, July, 2000, p. 113-124.

[17] Systolix Ltd (UK), product called PulseDSP technology has some similarities to FPGA architectures. Instead of using arrays of configurable logic blocks, it uses arrays of simple processors, each with its own memory and communications (EETimes, April 21,2000; http://www.systolix.co.uk/indexf.htm).

[18] Van Swaaij, M. , Catthoor, F., and DeMan, H., "Nonlinear Transforms for High Level Regular Array Synthesis: A Case Study," J. VLSI Signal Processing, vol. 4., pp. 259-268, 1992.

[19] Wilde, Doran K. and Sie, Oumarou, "Regular Array Synthesis using Alpha", Proc. 1994 Internation Conference on Application Specific Array Processors, IEEE Computer Press, p.200-211.

[20] Xilinx-new Virtex-II chips containing arrays of embedded multipliers: http://www.xilinx.com/products/virtex/