ASIC/FPGA CAD Tool for Automated Systolic Algorithm Mapping

J. Greg Nash

Centar (www.centar.net)
Los Angeles, California, USA

Abstract

A specialized ASIC/FPGA CAD tool is described that will take a user's high level code description of an algorithm and automatically generate abstract latency-optimal systolic arrays. Several new systolic mapping examples of the Lyapunov matrix equation (find *X*, given *AX+XB=C)* obtained using this CAD tool are described.

Introduction

A systolic implementation of an algorithm results a parallel architecture that is fine-grained and regular, and that supports local pipelined movement of data (1). Algorithms in this class have been shown to solve a large range of structured problems (e.g., linear algebra, graph theory, computational geometry, number-theoretic algorithms, string matching, sorting/searching, dynamic programming, discreet mathematics) (1)-(4). However, systolic arrays are still very difficult to design because a considerable knowledge of algorithms, architectures, and hardware is required to be successful and no related CAD tools are available.

In this paper a CAD tool, the symbolic parallel algorithm development environment (SPADE), is described which allows 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. It is applied to finding a variety new optimal systolic algorithm mappings of the matrix Lyapunov equation, which was chosen as a design example in part because it's systolic solutions represent a superset of a large fraction of previously published systolic solutions.

Related Work

Much research has been directed at finding systematic methodologies for finding optimal parallel implementations of recursive or iterative algorithms (1)-(4). Most of these deal with mathematical techniques, which, when applied to systems of "uniform" recurrence equations or their equivalent, result in parallel algorithms that represent "mappings" to an architectural model consisting of large arrays of locally connected virtual 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 of the PEs in the virtual array (5).

The disadvantage of tool methodologies based on uniform dependencies is that many important algorithms are not naturally expressed in this form and the process of putting them in this form can involve substantial effort. Furthermore, a particular choice of this form inherently restricts subsequent choice of architectures and thus the generated systolic arrays may not be optimal.

A better algorithm representation would be that of systems of non-uniform affine recurrence equations. An example is the matrix Lyapunov problem (7): find the solution to *AX+XB=C,* where *A* is an NxN non-singular lower triangular matrix, *B* is an NxN non-singular upper triangular matrix, *C* is an NxN matrix and *X* is an NxN unknown. The matrix equation can be represented in the equivalent form

$$c[i,j]=\sum_{k=1}^{i} a[i,k]*x[k,j]+\sum_{m=1}^{j} b[m,j]*x[i,m],$$

which leads directly by induction to the solution

$$\text{for } 1\le i,j\le N$$

$$x(i,j)=\frac{c(i,j)-\sum_{k=1}^{i-1} a(i,k)*x(k,j)-\sum_{m=1}^{j-1} b(m,j)*x(i,m)}{a(i,i)+b(j,j)}. \qquad (1)$$

This is a non-uniform recurrence equation and represents the form desired as an input to a CAD tool.

In order to solve the general mapping problem of non-uniform affine recurrence equations when desirable architectural constraints are imposed, it is necessary to use a search-based approach (6). The search methodology used by SPADE is based on (7), but involves a different formalism, is organized to provide better coverage of the architectural solution space for cases where solutions are less architecturally constrained, does additional analysis of potential solutions to give the designer more control over design tradeoffs, and includes a simulator that uses an embedded model of computation.

SPADE Description

A. Non-uniform Affine Recurrence Equations

A system of non-uniform affine recurrence equations is

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

where *f,g* represent the functional variable dependencies, I_j is the index range for equation *j*, w_i is one of algorithm variables and the affine indexing functions are *A(I)=AI+ a* and *B(I)=BI+b*. Here, *A/a,* and *B/b* are integer matrices/vectors. All assignments of values to variables 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 x from (1), $T(x) = T_x(A_x I + a_x) + t_x$, where T_x is a matrix and t_x is a vector. Thus, every variable *x(i,j)* gets mapped to a unique point in the space-time domain. For the example in (1) three indices specify *I*, so the mapping *T* for each variable is to one time dimension and two space dimensions, where the spatial index corresponds to a position in a virtual array of PEs.

Therefore, 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

$$T_x = \begin{bmatrix} \Lambda_x \\ S_x \end{bmatrix}, \qquad t_x = \begin{bmatrix} \gamma_x \\ s_x \end{bmatrix} \tag{3}$$

means that variable $x(i, j)$ 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 algorithm latency is the total number of these time steps needed to compute the entire result.

B. Solution Search

From (3) 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. Following (7), the allocation matrices like S_x are treated as a single variable, each derived from 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 and their possible values, SPADE examines all possible combinations and chooses those having 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 is computationally infeasible. This is solved (7) 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 (1) it can be seen that computation of x depends upon input a. Thus, it must follow temporally that

$$\Lambda_x(A_x I + a_x) + \gamma_x - \Lambda_a(B_a I + b_a) - \gamma_a \geq 0 . \tag{4}$$

Given the large number of dependencies seen in (1) this generates a large number of such constraints.

C. Input and output

Input to SPADE is in the form of high-level code based on a subset of the Maple scientific 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) that SPADE supports. For example, the Lyapunov algorithm can be written in this language directly from (1) as

```
for i to N do
 for j to N do                                                    (5)
   x[i,j]:=(c[i,j]-add(a[i,k]*x[k,j],k=1..i-1)-
      add(b[m,j]*x[i,m],m=1..j-1))/(a[i,i]+b[j,j])
 end do;
end do;
```

where the Maple "add" construct directly replaces the mathematical summation sign.

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.

Other inputs pertain to architectural constraints desired and objective function criteria used to select solutions from the search space. Architectural constraints specify which 2-D variables should be constrained to align with the time axis (normal to the variable plane in space-time is perpendicular to the time axis) and selection criteria picks out minimum latency solutions with (1) minimum bounding-box area, (2) maximum regularity and (3) minimum array bandwidth as secondary objective functions.

SPADE's primary outputs are the values *T/t* for each of the algorithm variables and a set of vectors that specify the direction of data flow for each dependency in the algorithm. For the example (1) it can be seen that x needs input from a, so corresponding to this dependence is a uniform flow of data from *T(a)* to *T(x)* in the space-time domain. Since this flow is one dimensional, a vector v_{xa} is calculated to indicate its direction. Consequently, in space-time *T(a)* represents a source of a data element moving in direction v_{xa} to a corresponding point at *T(x)*. 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 x.

Because the mathematical specification of *T* provides little insight into the nature of the solution, especially from the designer's point of view, 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 variable dependencies.

Lyapunov Algorithm Mapping

A. Minimum Area Designs

In this first mapping example the code (5) is supplied directly to SPADE as input. After parsing and analysis, two unique mappings were found, each with a latency of 4N-3 and area N(3N-1). (With *x*,*a* and *b* constrained to be "time aligned" two solutions with latency 6N-5 and area N(2N-1) were found in (7)).

The automatically generated graphical space-time mapping of one of these solutions is shown in Fig. 1. (In Fig.1 *IM1[i,j,k]=a[i,k]*x[k,j]* and *IM2[i,j,m]=b[m,j]*x[i,m]*, are two new variables created automatically in SPADE.)

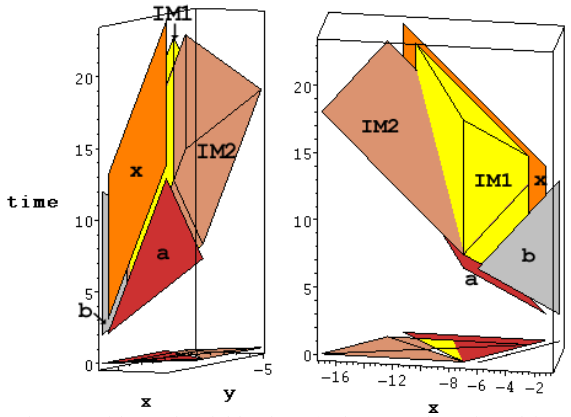

Fig. 1. Positions of variables from (1) in space-time viewed from two different perspectives (N=6) .

In Fig. 2 are the corresponding tool-generated projected view of the spatial array of PEs (one PE at each grid point) for each solution; superimposed on this are the various data-flows. The shaded regions are labeled in correspondence with the placements of the original algorithm variables, *x*, *a* and *b*. Since *c* has the same affine dependency as *x*, it is placed by SPADE (optionally) in the same locations as the variable *x*. Note that in the shaded lines (time-aligned variables, i.e., *b* and *x*) the normal of the "plane" of data is perpendicular to the time axis, so that there is a problem size amount of memory, O(N) per PE. SPADE has switches that allow the user control of alignment and placement of such planes.

As can be seen in Fig. 2, the PE arrays are non-uniform spatially and support nine different data-flows associated with the various algorithm variable dependencies. Some PEs experience data flow in five different directions. The picture in Fig. 2 shows all data flow paths. The actual time variation of data flow is more complex, with the size of uniform sub-regions of PEs growing and shrinking with time.

SPADE also provides options to explore sub-optimal latency solutions. This is very important when useful designs have different but closely spaced latencies. For example, if SPADE is set to explore minimum area designs for latencies of duration 4N-2, four unique solutions are found with smaller areas N(N+1) as shown in Fig. 3.

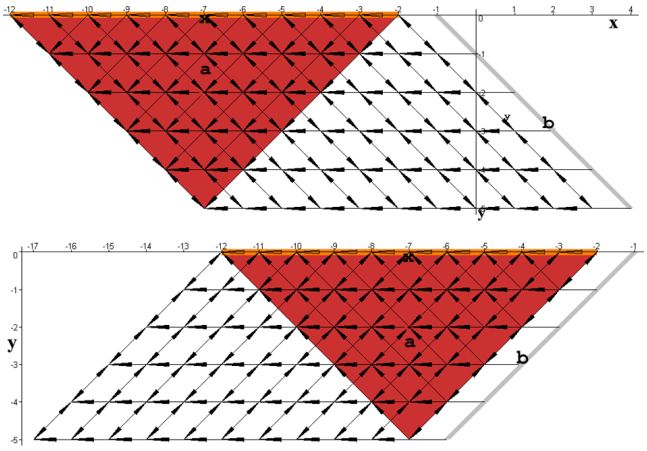

Fig. 2. Spatial projection (systolic array) and data flow for two area optimal space-time mappings having 4N-3 latency. Bottom design corresponds to space-time view in Fig. 1. (N=6)

B. Maximum Regularity Designs

ASIC/FPGA implementations are desired to be regular, preferably consisting of tiled blocks of memory, logic and connections. Consequently, a more regular systolic structure with a larger abstract area might be ultimately easier to implement. Therefore SPADE provides another secondary objective function that measures array regularity. The regularity metric scores a systolic array highest when

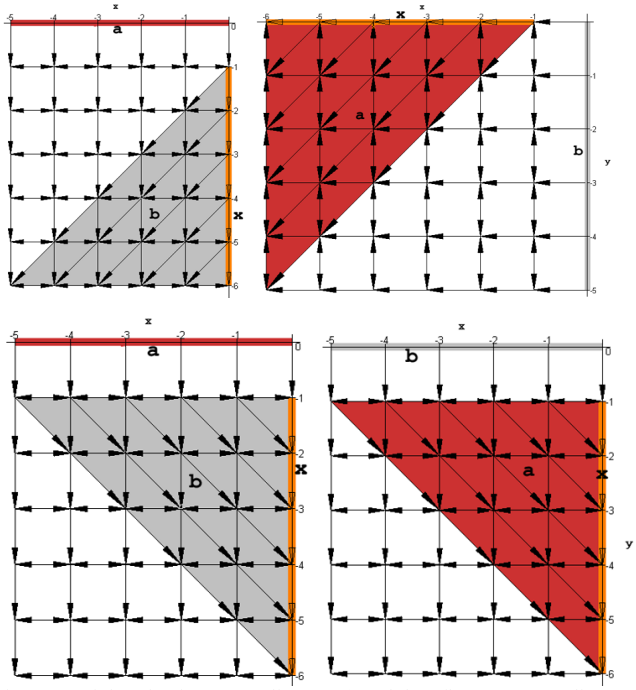

Fig. 3. Spatial projections (systolic arrays) and data flow corresponding to four area optimal space-time mappings with 4N-2 latency (N=6).

data flow occurs along orthogonal directions, when data flow paths are fewest in number, and when inputs and outputs occur as close together as spatially possible. SPADE penalizes designs under this criteria that result in a

plane of data is that is constrained along a line, as *b* in Fig. 2, and when this line is not on the boundary of the array.

Looking at all designs having a latency of 4N-2, four unique designs are found, each having an area (N+1)(N+1), which is close to the minimum area designs found in Fig. 3. In this case algorithm variables *a* and *b* appear only at the array edge, although *x* is now distributed across the array. (Occasionally it may be desirable for an output to remain resident in an array after systolic processing, in particular if there's another phase of processing involved.) All of these designs have the same systolic architecture and PE connectivity shown in Fig. 4. However, each of the four designs have different data flowing among these connections (this can only be seen by viewing the corresponding space-time views).

Calculated values of *T,t* and *v* for some of the designs described above is provided in Table 1.

Summary

Although this tool can be used for ASIC design, FPGAs conceptually provide a better implementation strategy for systolic arrays. Here, a designer faces a large number of design options, given the variety of reconfigurable computers and related hardware. 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 because using SPADE moves 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. A simulator is available that has an embedded computational model to make the transition to systolic hardware more direct.

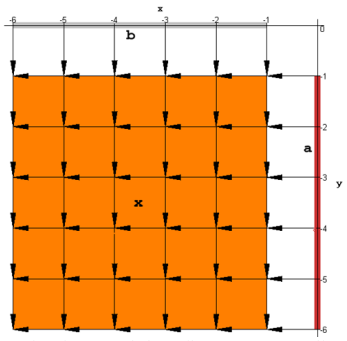

Fig. 4. Spatial projections and data flow corresponding for maximum regularity systolic array with latency 4N-2 (N=6).

Acknowledgements

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

References

(1) Kung, S.Y., "VLSI Array Processing", Prentice Hall, 1988.

(2) D. I. Moldevan, "Parallel Processing", Morgen Kaufmann, 1993.

(3) Keshab K. Parhi, "VLSI Digital Signal Processing Systems", John Wiley, 1999, Chapter 7.

(4) P. Quinton, and Y. Robert, "Systolic Algorithms and Architectures", Prentice Hall 1991.

(5) 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. 113-124b.

(6) Paul Feautrier, "Fine Grain Scheduling under Resource Constraints," 7th Workshop on Language and Compilers for Parallel Computers, Aug. 1994.

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

Table 1. Transformation matrices*T,t* and data flow vectors, *v*, for designs in Figs. 2, 3 and 4.

	x	a	b	IM1	IM2	v_{xa}	v_{xb}	v_{xIM1}	v_{xIM2}	v_{IM1a}	v_{IM1x}	v_{IM2b}	v_{IM2x}
Fig. 2 (top)	$\begin{bmatrix}2&2\\-2&0\\0&0\end{bmatrix},\begin{bmatrix}-1\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\-1&-1\\-1&1\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\-1&1\\1&-1\end{bmatrix},\begin{bmatrix}0\\-1\\0\end{bmatrix}$	$\begin{bmatrix}1&2&1\\-1&0&-1\\-1&0&1\end{bmatrix},\begin{bmatrix}-1\\0\\0\end{bmatrix}$	$\begin{bmatrix}2&1&1\\-2&1&-1\\0&-1&1\end{bmatrix},\begin{bmatrix}-1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}-2\\0\end{bmatrix}$	$\begin{bmatrix}-1\\1\end{bmatrix}$	$\begin{bmatrix}-1\\1\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}-1\\-1\end{bmatrix}$	$\begin{bmatrix}-2\\0\end{bmatrix}$	$\begin{bmatrix}1\\-1\end{bmatrix}$
Fig. 3 (top left)	$\begin{bmatrix}2&2\\0&0\\0&-1\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\-1&1\\0&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\1&-1\\0&-1\end{bmatrix},\begin{bmatrix}1\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&2&1\\-1&0&1\\0&-1&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}2&1&1\\0&-1&1\\0&-1&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	$\begin{bmatrix}-1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}-1\\-1\end{bmatrix}$
Fig. 4	$\begin{bmatrix}2&2\\0&-1\\-1&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\0&0\\0&-1\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&1\\0&-1\\0&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1&2&1\\0&-1&0\\0&0&-1\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}2&1&1\\0&-1&0\\-1&0&0\end{bmatrix},\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}-1\\0\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}-1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\end{bmatrix}$	$\begin{bmatrix}0\\-1\end{bmatrix}$	$\begin{bmatrix}-1\\0\end{bmatrix}$