OPTIMAL ALGORITHM MAPPING FOR FAST SYSTOLIC ARRAY IMPLEMENTATIONS

Igor Ozimek

Institute Jožef Stefan, Ljubljana, Slovenia

Key words: systolic arrays, parallel algorithm mapping, microcycled dependence graph (μDG), optimal scheduling, loop-extraction algorithm (LEA), VLSI

Abstract: There are a number of algorithms which are described by a set of recursive equations of regular dependences. Examples are certain filtering algorithms. These algorithms can be efficiently mapped to the systolic array structure, which can then be implemented in VLSI technology. This paper deals with the problem of finding optimal scheduling for a given algorithm, taking into account its exact computational requirements. First we introduce microcycled Dependence Graph (DG) and, using the notion of microcycles, define the speed of its execution that has to be maximised. Then, using Reduced Dependence Graph (RDG), we express the upper bound on the computation speed as a set of inequalities defined by the loops in RDG. To find these loops, we define a Loop Extraction Algorithm (LEA). Solving the set of inequalities obtained does not conform exactly to the linear programming problem. We describe a procedure that makes it possible to use the linear programming method to find the optimal scheduling vector.

Optimalna preslikava algoritmov za hitro izvajanje v sistoličnem polju

Ključne besede: sistolična polja, vzporedne preslikave algoritmov, mikrokoračni graf odvisnosti, optimalno časovno razvrščanje, algoritem odkrivanja zank, VLSI

Izvleček: Mnogo algoritmov lahko zapišemo kot sistem rekurzivnih enačb z regularnimi odvisnostmi. Primer so razni filtri. Take algoritme lahko učinkovito preslikamo v sistolična polja, ki jih lahko nato izvedemo v tehnologiji VLSI. Ta prispevek se ukvarja s problemom iskanja optimalnega časovnega razvrščanja računskih operacij danega algoritma z natančnim upoštevanjem njegovih računskih zahtev. Najprej vpeljemo mikrokoračni graf odvisnosti in z uporabo pojma mikrokoraka določimo hitrost izvajanja, ki naj bo čim večja. Potem z uporabo reduciranega grafa odvisnosti izrazimo zgornjo mejo hitrosti računanja kot sistem neenačb, ki jih določajo zanke v reduciranem grafu odvisnosti. V ta namen določimo algoritem odkrivanja zank. Dobljeni sistem neenačb ne ustreza povsem postopku reševanja z metodo linearnega programiranja. S pomočjo dodatnega postopka omogočimo uporabo te metode za določitev optimalnega vektorja izvajanja algoritma.

1. Introduction

Certain real-time applications, such as signal filtering and processing in a digital communication system, require the use of a special, massively parallel computing structure called the systolic array structure to achieve acceptable performance. To implement an algorithm in this way we need a mapping procedure to map the set of equations, which describe the algorithm, to a systolic array. This mapping consists of scheduling (i.e. time mapping, mapping each DG node to a particular time instant) and space mapping (mapping each DG node to a systolic array cell). The methods described in the literature /1,2,3,4,5/ are best suited for simplified systems of equations that consist of one main equation, which describes the algorithm, and a number of auxiliary equations, which are used to achieve the local communication and single assignment properties.

In this paper we consider the problem of scheduling, and develop a new approach to find the optimal scheduling of complicated algorithms described by a set of equations which have to fulfil the requirement of regularity, i.e. constant dependence vectors. Our procedure takes into account the exact computational requirements of the basic arithmetic operations used, and yields an optimal scheduling vector that guarantees the fastest possible computation of the algorithm. Space mapping can then be accomplished using methods known from the literature /2/.

2. DG and microcycled DG

DG (Dependence Graph) is one of the basic tools in the systolic array mapping process. To describe it and its modification, μDG (*microcycled DG*), we shall take a simple example of matrix-vector multiplication:

$$\mathbf{c} = \mathbf{A}\mathbf{b} \tag{1}$$

Eq. (1) can be written as:

$$c_i = \sum_{j=1}^{N} a_{ij} b_j \tag{2}$$

or, recursively, as:

$$c_i = c_i + a_{ij} b_j \tag{3}$$

To be executed by a systolic array, Eq. (1) must be transformed to its equivalent recursive form in such a way that broadcasting (variable *b* in Eq. (3)) and multiple assignment (variable *c* in Eq. (3)) are eliminated:

$$b_{i+1,j} = b_{i,j}$$
$$c_{i,j+1} = c_{i,j} + a_{i,j}b_{i,j} \quad (4)$$

where the first equation is used to eliminate broadcasting of variable b (for details see /2/).

2.1. Dependence Graph

The corresponding DG of Eq. (4) for the case of a 3×3 matrix **A** and 3×1 vectors **b** and **c** is shown in Fig. 1. Variables *i* and *j* are indices of the algorithm and DG nodes. Each DG node represents one iteration of (repetitive) calculations needed by the algorithm. In our case an iteration consists of a multiplication (*ab*), an addition (*c*+...), and a shift-through (variable b). Each edge of DG represents a dependence between individual iterations. Since there are no multiple assignments, DG does not have any loops. Since there is no variable broadcasting, dependence vectors are local. In addition, as a prerequisite, the given algorithm is regular, i.e. its execution is independent of the indices *i* and *j*. Thus, DG is regular, localised and without loops, and is as such suitable for mapping to the systolic array structure.

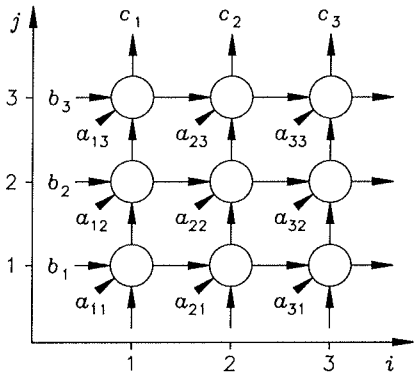

Fig. 1. DG for Eq. (4)

Scheduling for a regular DG can be represented by equitemporal lines (planes for 3-D DG or hyperplanes for multi-D DG). In Fig. 2, a simplified DG from Fig. 1 is shown together with a possible scheduling. The scheduling vector s is defined as having components equal to the number of equitemporal lines between neighbouring nodes in the corresponding directions. Needless to say, these values are integers. For Fig. 2, $\mathbf{s} = [1,1]^T$. By this definition, a scheduling vector is orthogonal to equitemporal lines and its size is proportional to the *slowness* of computing. Thus, the smaller s becomes, the better.

The execution time index of a particular DG node can be expressed as:

$$t = \mathbf{s}^T \begin{bmatrix} i \\ j \end{bmatrix} \quad (5)$$

which is true also for the μDG described below.

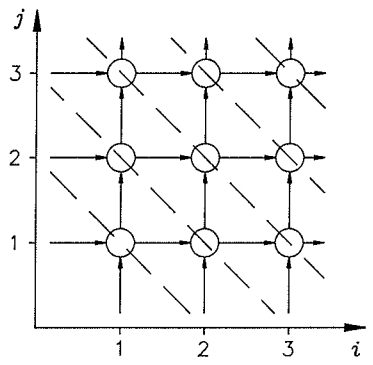

Fig. 2. Simplified DG for Eq. (4) with scheduling

2.2. Microcycled Dependence Graph

Scheduling in Fig. 2 does not take into account the real computation requirements within nodes. We can consider the execution within a node to be performed in microcycles and hidden from the outside world, while DG (and its corresponding systolic array) shifts data between nodes (cells) in macrocycles. These shifts can take place only after a complete (microcycled) computation within a node is finished. It can be shown that this approach is suboptimal. For our purpose it will suffice to take all three operations (multiplication, addition and shift-through) as being of equal complexity, i.e. requiring one microcycle each to execute. The main (second) equation in Eq. (4) involves a multiplication and an addition in sequence, thus requiring two microcycles. The first equation of Eq. (4) can be executed in parallel, requiring only one microcycle. There are 5 macrocycles needed for execution of DG in Fig. 2, so the total number of required microcycles is 10.

A better solution can be found using μDG. It eliminates the notion of macrocycles and looks at DG entirely in terms of microcycles. In Fig. 3, μDG is shown which corresponds to DG in Fig. 2. The black dots along the edges and within the nodes correspond to the basic arithmetic operations that can be executed within one microcycle. For the systolic array mapping, the actual execution is placed into the node to which the corresponding dependence edge is directed. Fig. 4 illustrates this by showing a node with its related operations indicated by the shaded area.

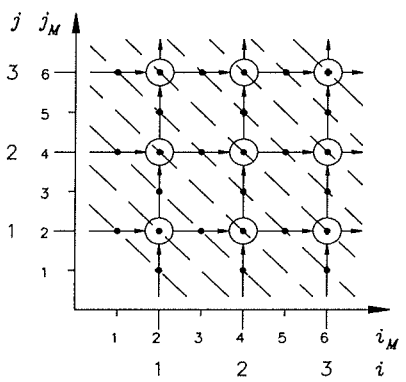

Fig. 3. DG for Eq. (4)

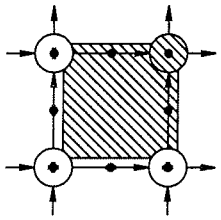

Fig. 4. A node with its related operations

There are two types of indices in Fig. 3. Indices i and j are node indices, which are the same as above, while i_M and j_M are microindices, related to our proposed microcycled scheme. The relation between them is:

$$\begin{aligned} i_M &= s_i i \\ j_M &= s_j j \end{aligned} \tag{6}$$

where s_i and s_j are the components of scheduling vector s:

$$\mathbf{s} = \begin{bmatrix} s_i \\ s_j \end{bmatrix} \tag{7}$$

Using μDG, a better scheduling for the algorithm described by Eq. (4) can immediately be found. Since in the i direction only one cycle is needed (propagation of variable b), scheduling can be as in Fig. 5, $\mathbf{s} = [2,1]^T$, requiring 8 microcycles for complete execution.

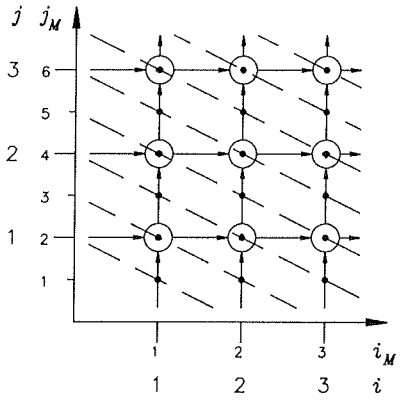

Fig. 5. A better scheduling for Eq. (4)

It can readily be shown that even the scheduling in Fig. 5 is not optimal. The optimal one is shown in Fig. 6 and is achieved by using a pipelined computation path between variables b and c. The relationship between the two variables is shown in Fig. 7. μDG in Fig. 6 needs a total of 5 microcycles to execute, but since the b and c variable planes are shifted relative to each other due to pipelining, 6 microcycles are actually needed, outperforming our initial solution of 10 microcycles by factor of almost 2.

In the sequel, a formal procedure will be developed for finding the optimal scheduling vector of a given algorithm.

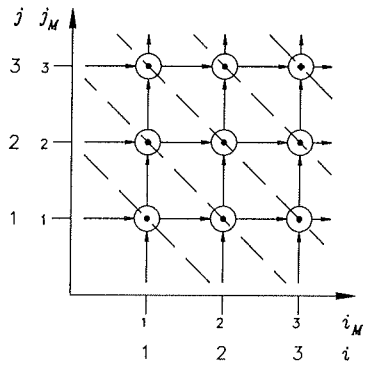

Fig. 6. The optimal scheduling for Eq. (4)

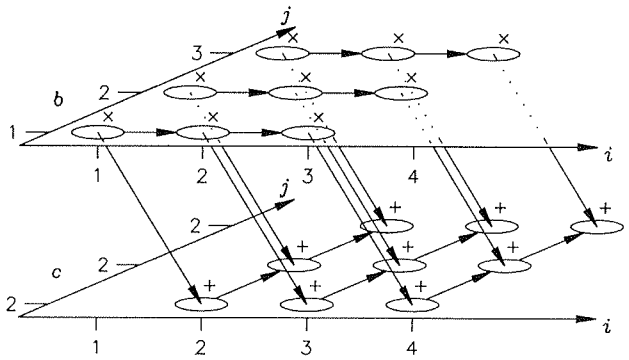

Fig. 7. Relationship between variable b and c planes of DG

3. Finding the optimal scheduling

In this section we first describe RDG (*Reduced Dependence Graph*), /3/, and show that the maximum speed of computation is limited by the loops in RDG. Then we propose an algorithm for automatic extraction of RDG loops. In this way we obtain a system of inequalities which define the space of possible scheduling vectors $\mathcal{S}$. We then use the linear programming method, with some extensions, to find the optimal scheduling vector **s**.

3.1. RDG - Reduced Dependence Graph

RDG is a graph that has a node for every variable of the given algorithm and an edge for every dependence between them. Its name (*reduced*) comes from the fact that, contrary to DG, nodes and dependences are not repeated n times but appear only once.

Two data belong to each edge e_k,: the *dependence vector* $\mathbf{d}_k$, denoting the distance in μDG between the input and output variables, and *computational complexity* r_k, denoting the number of microcycles required to compute the output variable from the input variable.

RDG for our matrix-vector multiplication example is shown in Fig. 8.

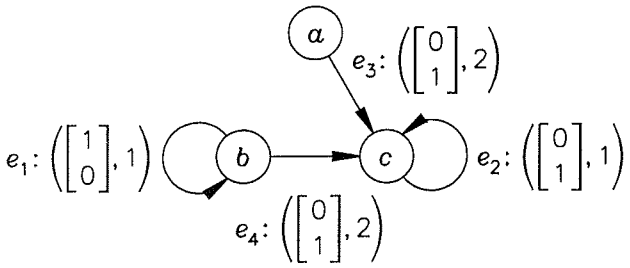

Fig. 8. RDG for Eq. (4)

The dependence vectors for RDG in Fig. 8 are:

$$\mathbf{d}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{d}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{d}_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{d}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{8}$$

and the corresponding computational complexities are:

$$r_1 = 1, \quad r_2 = 1, \quad r_3 = 2, \quad r_4 = 2 \tag{9}$$

Alternatively, we can write dependences and computational complexities in the matrix and vector forms respectively:

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{10}$$

$$\mathbf{r} = \begin{bmatrix} 1 & 1 & 2 & 2 \end{bmatrix} \tag{11}$$

3.2. RDG loops and the speed of computation

The fastest computation of an algorithm is defined by the loops of its RDG. A loop describes the computation of an instance of a variable on the basis of its previous instance(s). This is ultimately a sequential process, which limits the maximum speed of computation. Let us illustrate this by a simple example of the next circularly dependent algorithm:

$$\begin{aligned} a_{i,j} &= f_a(b_{i-1,j}) \\ b_{i,j} &= f_b(c_{i-1,j-1}) \\ c_{i,j} &= f_c(a_{i,j-1}) \end{aligned} \tag{12}$$

Functions *f* denote arbitrary arithmetic operations. The corresponding RDG is shown in Fig. 9.

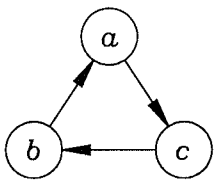

Fig. 9. RDG for Eq. (12)

Fig. 10 shows the corresponding DG, split to its three variable planes. Computation of variable *b* depends on one of its previous values, which is located $[2,2]^T$ back in DG. ("Previous" here has a somewhat special meaning, since no time is assigned to the computations at this stage of the mapping process - indices i and j are not yet related to the final space or time indices. Naturally, a result that is used further in another computation has to be computed earlier in time.) On this computation path (showed as a dashed line in plane *b*) there must be enough microcycles available to perform all the necessary arithmetic operations. Since the number of available microcycles depends on the scheduling vector **s**, this represents the lower bound on s or, more exactly, on its components.

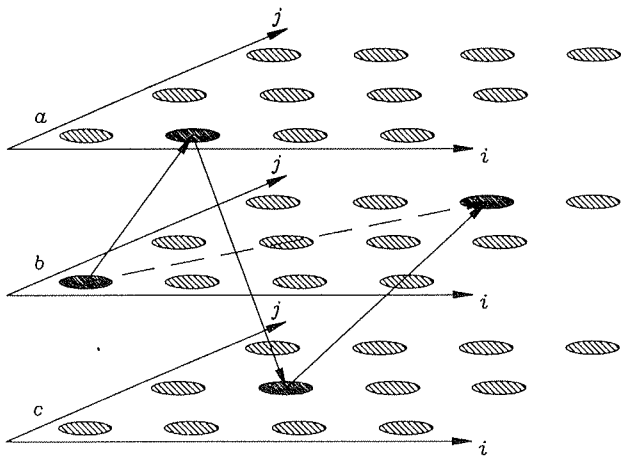

Fig. 10. DG for Eq. (12), split to variable planes

The fastest computation can be determined by finding and taking into account all the loops. For each (*l*-th) loop, two additional data are computed: the cumulative dependence vector, $\mathbf{d}_{L,l}$, and the computational complexity of the loop, $r_{L,l}$. They are calculated as the sum of the dependence vectors and the sum of the computational complexities, respectively, of all the edges belonging to the loop.

The number of available microcycles along the *l*-th loop equals $\mathbf{s}^T\mathbf{d}_{L,l}$. The scheduling vector **s** must therefore satisfy the following set of inequalities:

$$\mathbf{s}^T\mathbf{d}_{L,l} \geq r_{L,l} \quad \text{for all } l \tag{13}$$

or, in the matrix form:

$$\mathbf{s}_L^T\mathbf{D}_L \geq \mathbf{r}_L \tag{14}$$

If we return to our matrix-vector multiplication example, we can easily find two loops. Their corresponding dependences and computational complexities are:

$$\mathbf{D}_L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{15}$$

$$\mathbf{r}_L = \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{16}$$

The resulting optimal scheduling vector **s**, which satisfies (14) with equality, is:

$$\mathbf{s} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{17}$$

For our simple matrix-vector multiplication example it has been straightforward, but for a more general case neither finding the loops of a RDG nor determining the optimal scheduling vector from them is a trivial task. To illustrate its complexity it is enough to take a look at the RLSL (*Recursive Least Square Lattice*) algorithm /6/ in Table 1 and its corresponding RDG in Fig. 11. Obviously we need a more powerful approach to tackle such problems.

Prediction recursions:

$$\Delta_{i-1,j} = \lambda\Delta_{i-1,j-1} + \frac{b_{i-1,j-1} f_{i-1,j}}{\gamma_{i-1,j-1}}$$

$$\Gamma^{f}_{i,j} = -\frac{\Delta_{i-1,j}}{\mathcal{B}_{i-1,j-1}}$$

$$\Gamma^{b}_{i,j} = -\frac{\Delta_{i-1,j}}{\mathcal{F}_{i-1,j}}$$

$$f_{i,j} = f_{i-1,j} + \Gamma^{f}_{i,j} b_{i-1,j-1}$$

$$b_{i,j} = b_{i-1,j-1} + \Gamma^{b}_{i,j} f_{i-1,j}$$

$$\mathcal{F}_{i,j} = \mathcal{F}_{i-1,j} - \frac{\Delta^2_{i-1,j}}{\mathcal{B}_{i-1,j-1}}$$

$$\mathcal{B}_{i,j} = \mathcal{B}_{i-1,j-1} - \frac{\Delta^2_{i-1,j}}{\mathcal{F}_{i-1,j}}$$

$$\gamma_{i,j-1} = \gamma_{i-1,j-1} + \frac{b^2_{i-1,j-1}}{\mathcal{B}_{i-1,j-1}}$$

JPE recursion:

$$\rho_{i,j} = \lambda\rho_{i,j-1} + \frac{b_{i,j}}{\gamma_{i,j}} e_{i,j}$$

$$\kappa_{i,j} = \frac{\rho_{i,j}}{\mathcal{B}_{i,j}}$$

$$e_{i+1,j} = e_{i,j} - \kappa_{i,j} b_{i,j}$$

Table 1. RLSL recursions

3.3. LEA - Loop Extraction Algorithm

To find the loops of an RDG, we propose the loop-extraction algorithm described in Table 2. With it, a number of directed trees are built. Their nodes represent the variables of the given system. Their (towards the root directed) edges represent inter-variable dependences. The following additional data belong to each edge: *the dependence vector*, denoting the distance in μDG between the input and output variables, and *computational complexity*, denoting the number of microcycles required to compute the output variable from the input variable. Trees are built from the roots to the leaves. Each path is built until either a node is repeated or a dead end is reached. Since any repetition of a node in a path stops its growth, the maximum tree depths are equal to the number of the variables.

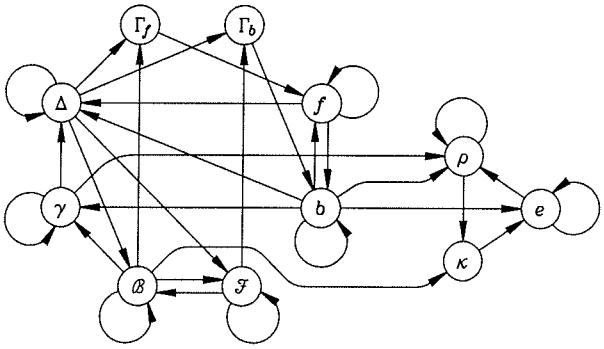

Fig. 11. RDG for RLSL (Table 1)

1. Make a list of all non-processed nodes (*all_nodes*) and a list of all possible roots (*all_roots*). At the beginning, both lists are equal and consist of all the variables of the given system of equations.

2. Begin with the first (*i*=1) TCC (*Tightly Connected Component*, /3/).

3. For the *i*-th TCC, create two empty lists: a list of its roots (*roots(i)*) and a list of its loops (*loops(i)*). Move the first node from *all_roots to roots(i)*.

4. From *roots(i)* take the first node (delete it from the list) and build a tree as described previously, but using only the nodes from *all_nodes*.

5. Delete the node just taken from *roots(i)* from *all_nodes*. This serves mainly to reduce the algorithm complexity by eliminating repetitive loop generation.

6. Each path in the tree with the ending leaf node equal to the root node represents a loop of the current (*i*-th) TCC. Add it to *loops(i)*.

7. Add to *roots(i)* every leaf that is not equal to the root node but is equal to some previous node on the same paths (i.e. the node is repeated) and is at the same time used in at least one loop in *loops(i)*. (These nodes have their own loops, not containing the current root node. They will be used later to build more trees for the current TCC.)

8. If *roots(i)* is not empty, go to 4.

9. The current (*i*-th) TCC processing has just been done. From *all_roots* delete all the nodes that are used in any loop in *loops(i)*. If *all_roots* is not empty, choose the next TCC (*i.e.: i←i+1*) and go to 3.

10. The processing is done. The result is a set of TCCs with corresponding sets of loops (*loops(i)*).

Table 2. LEA algorithm

The system of inequalities, Eq. (14), can be reduced by eliminating multiple dependence vectors of the same magnitude. From each set of identical dependence vectors with different computing complexities, only the highest computational complexity is used.

An example of LEA execution for the case of the RLSL algorithm from Table 1 is shown in Fig. 12.

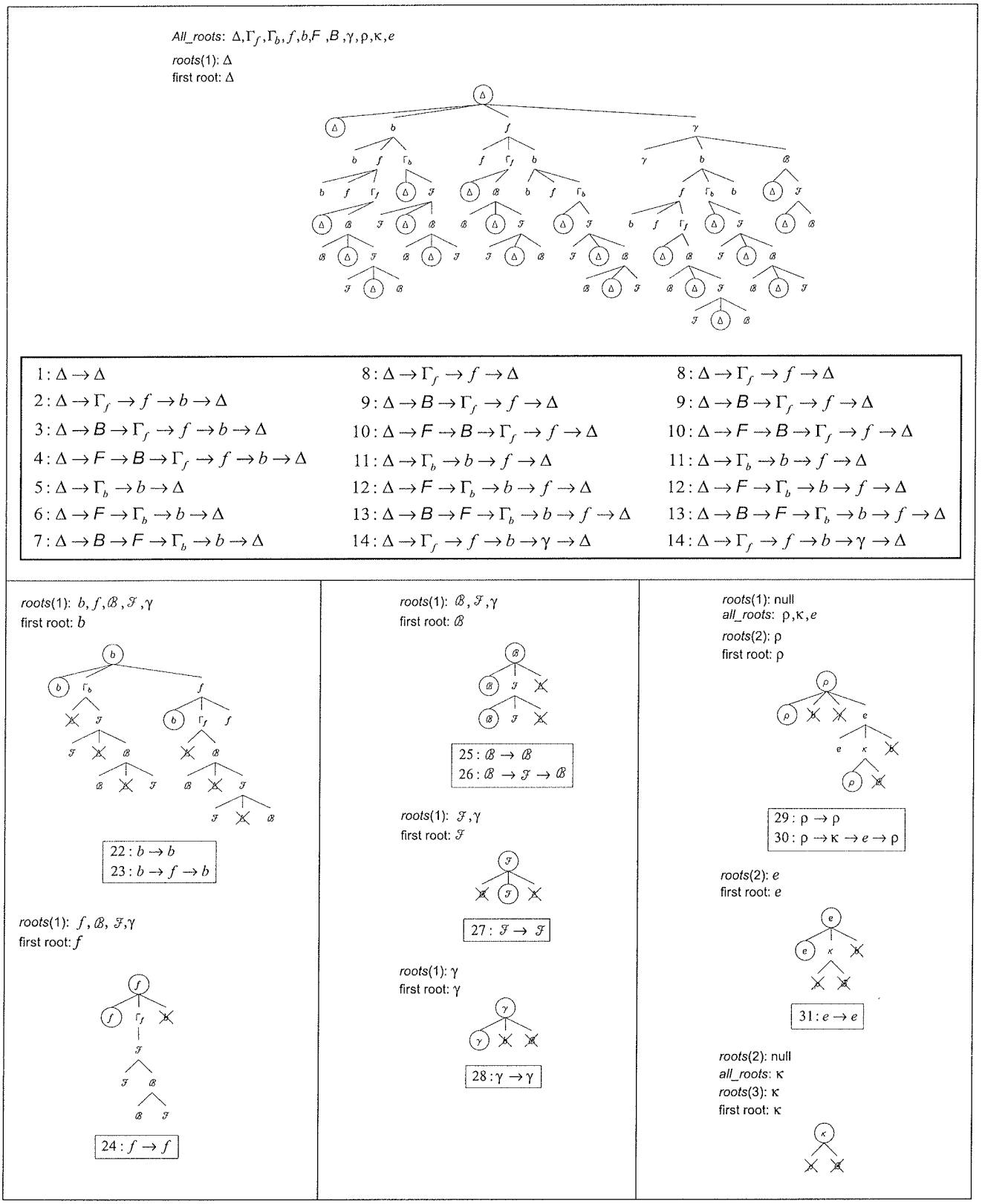

Fig. 12. LEA execution for RLSL

3.4. Space of possible scheduling vectors

In general, a system of inequalities like Eq. (14) need not have any solution. However, in the case of parallel algorithm mapping the corresponding DG is checked at the beginning for computability or causality, in the directions of infinite or finite dimension respectively. (For computability, no DG loops are allowed. For causality, all the edges must be oriented positively with respect to any infinite DG dimension.) Since DG is at least computable, a valid scheduling vector **s**, and hence at least one solution of the system of inequalities, must exist. Additionally, there is no upper limit to the size of **s** (i.e. the slowness of computing), providing that **s** is oriented in an appropriate direction. Eq. (14) defines the space of valid scheduling vectors $\mathcal{S}$. This space is similar to a polygon (for 2-D DGs, polyhedron for 3-D DGs), but is unbounded in certain directions towards infinity. Fig. 13 illustrates this by showing $\mathcal{S}$ spaces for four hypothetical algorithms with different dependences and computational complexities.

Given the space of possible scheduling vectors $\mathcal{S}$, the optimal scheduling vector **s** can be found. The optimal **s** is the one that guarantees the fastest computation of the given algorithm (the shortest or the fastest pass through its DG). For the case (a) in Fig. 13 the solution is unique, $\mathbf{s} = [2,3]^T$. The solutions to the other three cases in Fig. 13 are less obvious.

3.5. Finding the optimal scheduling vector

With some modifications, the problem of finding the optimal scheduling vector **s** in the space of possible scheduling vectors $\mathcal{S}$ can be made to conform to the requirements of the linear programming method.

3.5.1. Linear programming method (LPM)

The linear programming problem is defined by the following set of equations, /7/. Eq. (18) restricts the feasible region to the non-negative portion of R^N:

$$x_1 \geq 0, \ldots, x_N \geq 0 \tag{18}$$

Eq. (19) is the main system of inequalities, which further bound the feasible region:

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1N}x_N &\geq b_1 \\ \ldots \quad \ldots \quad \ldots \quad \ldots & \\ a_{M1}x_1 + \ldots + a_{MN}x_N &\geq b_M \end{aligned} \tag{19}$$

and Eq. (20) is the objective function that has to be minimised (or maximised):

$$f(x_1, \ldots, x_N) = c_1x_1 + \ldots + c_Nx_N \tag{20}$$

The above equations can be written in the matrix form:

$$\begin{aligned} \mathbf{x} &\geq \mathbf{0} \\ \mathbf{Ax} &\geq \mathbf{b} \\ f(\mathbf{x}) &= \mathbf{cx} \end{aligned} \tag{21}$$

3.5.2. Applying LPM to the optimal scheduling problem

The optimal scheduling problem differs from the linear programming problem at two points.

The first point is that the space $\mathcal{S}$ (defined by Eq. (13)) is not limited to the non negative portion of R^N. The problem can be solved by decomposing it to a number of subspaces, as in Fig. 14, so that none of them crosses the quadrant boundaries. LPM is then applied to each of these subspaces and the solutions are combined.

By decomposing R^N in the way described, we get 2^N subspaces, not all of which are occupied by $\mathcal{S}$. Since the dimensionality of algorithms is usually low (typically 2 or 3) the number of subspaces is not a problem.

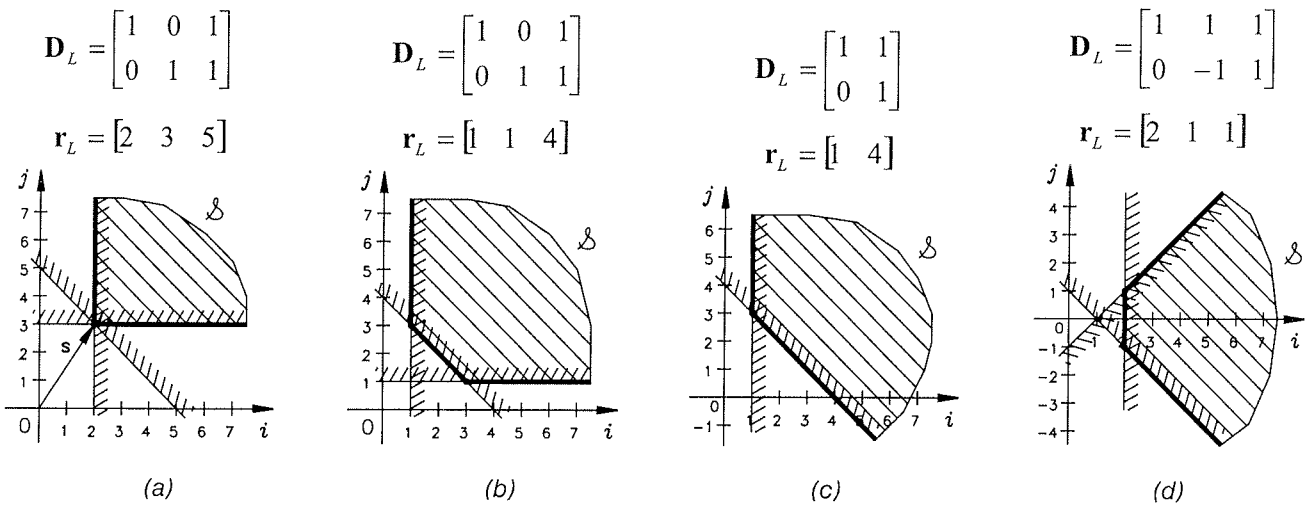

Fig. 13. Examples of 2-D $\mathcal{S}$ spaces

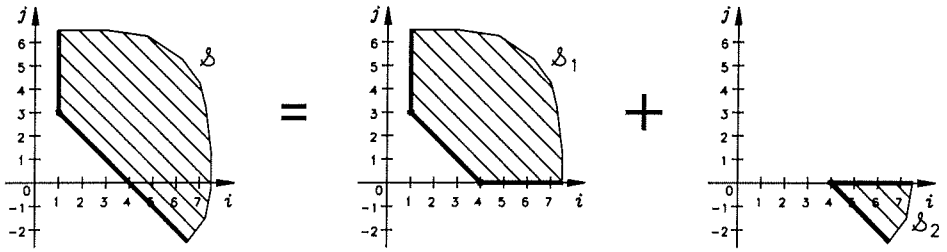

Fig. 14. Decomposition of $\mathcal{S}$ space

As we see, the feasible region for our problem is unbounded, but the problem itself is (a minimal **s** does exist). The reason for that is that, by definition, a possible scheduling vector exists (otherwise systolic array implementations would be impossible - which was checked for at the beginning of the mapping process), and that there certainly exists a lower bound for it (otherwise computing of the algorithm could be made arbitrarily fast with all DG nodes being computed in parallel, which is impossible due to the mutual dependences between them).

The second point of discrepancy between LPM and our scheduling problem is that the objective function for the scheduling problem, unlike Eq. (20), is not linear. For bounded DGs, the objective function can be defined as the number of cycles needed to traverse it:

$$f(s_1,\ldots,s_N)=\max_{1\le n\le N}(|s_n|)-1+\sum_{n=1}^{N}(l_n-1)|s_n| \quad (22)$$

where s_n is the n-th component of the scheduling vector **s**, and l_n is the size of DG in the n-th direction.

In the first quadrant (according to the decomposition described above) $s_n \ge 0$, and Eq. (22) takes the following form (for each of the quadrants the form is exactly the same, provided that the signs of variables are changed accordingly):

$$f(s_1,\ldots,s_N)=\max_{1\le n\le N}(s_n)-1+\sum_{n=1}^{N}(l_n-1)s_n,\ \text{for } s_n\ge 0 \quad (23)$$

where function *max* is the only non-linear term.

The cycles needed for the computation to traverse a DG can be represented by equitemporal lines drawn in the DG. Fig. 15 shows this for 3×3 DG for case (c) from Fig. 13. Equitemporal lines are shown for the scheduling vector values of (a): $[1,3]^T$, (b): $[2,2]^T$, (c): $[3,1]^T$, (d): $[4,0]^T$, and (e): $[5,-1]^T$. The optimal solution is (b).

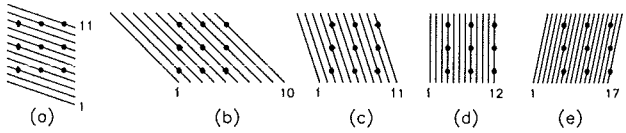

Fig. 15. Scheduling and the exact objective function Eq. (22)

By dropping the first term on the right side of Eq. (22) we obtain:

$$f(s_1,\ldots,s_N)=\sum_{n=1}^{N}(l_n-1)|s_n| \quad (24)$$

and Eq. (23) becomes linear, as required by LPM:

$$f(s_1,\ldots,s_N)=\sum_{n=1}^{N}(l_n-1)s_n,\quad \text{for } s_n\ge 0 \quad (25)$$

Eqs. (24) and (25) neglect those starting cycles that take place before the computation leaves the first DG node. The situation is illustrated by Fig. 16 and recapitulated in Table 3.

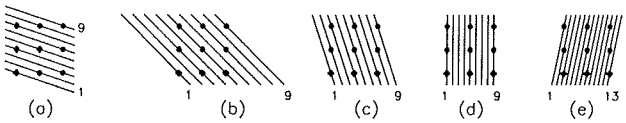

Fig. 16. Scheduling and the approximate objective function Eq. (24)

	s	Exact cycle no.	Approx. cycle no.
(a)	$[1,3]^T$	11	9
(b)	$[2,2]^T$	10	9
(c)	$[3,1]^T$	11	9
(d)	$[4,0]^T$	12	9
(e)	$[5,\ 1]^T$	17	13

Table 3. Exact and approximate objective functions

The relative error of the approximate objective function can be expressed as:

$$\varepsilon_r=\frac{\max_{1\le n\le N}(|s_n|)}{\max_{1\le n\le N}(|s_n|)+\sum_{n=1}^{N}(l_n-1)|s_n|} \quad (26)$$

which decreases towards zero when DG (coefficients *l*) grows towards infinity.

The objective function Eq. (22) has been defined for bounded DGs. A DG can be unbounded in a direction. This happens in the case of real time signal processing, where the input signal arrives as a continuous stream of data. The infinite dimension of DG corresponds to the time axis. For such a case the value of Eq. (22) is infinite, and the following simple objective function can be used:

$$f(s_1,\ldots,s_n) = s_u \qquad (27)$$

where s_u is the component of **s** corresponding to the unbounded direction, which is presumed to be positive. This objective function is, unlike Eqs. (22) or (24), both linear and exact.

3.5.3. Refining the LPM solution for optimal scheduling - sub-decomposition

The solution obtained by using the approximate objective function Eq. (24) may be sufficiently good, especially for large DGs. Even for our example of the very small 3×3 DG, the difference between the best approximate (true optimal) solution (b) and the worst approximate "optimal" solution (d) is only 20%.

If we nevertheless want to find the true optimal solution, we can do so by decomposing each subspace from section 3.5.2. in the following ways:

$$1: |i| \geq |j|$$
$$2: |j| \geq |i|$$

for the 2-D case,

$$1: (|i| \geq |j|) \wedge (|i| \geq |k|)$$
$$2: (|j| \geq |i|) \wedge (|j| \geq |k|)$$
$$3: (|k| \geq |i|) \wedge (|k| \geq |j|)$$

for the 3-D case, and so on.

In this way, each subspace from section 3.5.2. is further decomposed to N sub subspaces. Within each of these the exact objective function Eq. (23) becomes linear. For the first sub-subspace, Eq. (23) takes the following form:

$$f(s_1,\ldots,s_N) = s_1 - 1 + \sum_{n=1}^{N} (l_n - 1)s_n, \quad \text{for } s_1 \geq s_n \geq 0 \qquad (28)$$

LPM can then be applied, the solutions obtained combined together and, from them, the best one taken. Since the dimensionality of algorithms is usually small (e.g. N = 3) the number of subspaces to be processed is not so great as to constitute a problem.

3.5.4. Integer programming methods

The components of scheduling vector s are constrained to take only integer values. This requirement is not fulfilled by the general LPM, so integer programming methods have to be used (e.g. branch-and-bound algorithm, cutting plane algorithm) the details of which can be found in /8/ and /9/. Integer programming is an extension to LPM; if the original LPM solution does not satisfy the integer requirement, additional constraints are imposed to force an integer solution.

4. Conclusions

We have developed a procedure for finding the optimal scheduling for a systolic array implementation of an algorithm. The procedure has been manually tested on the RLSL algorithm from Table 1. In its RDG (Fig. 11), 31 loops have been found. The complexity of this problem is sufficiently great to make it difficult to handle without the procedures described above.

The need for sophisticated signal processing algorithms is increasing with the introduction of more and more complex communication systems. At the same time, VLSI technology is becoming capable of implementing complex computing hardware on a chip. The procedures described in this paper can be used as a tool for designing such specialised VLSI circuits.

References

/1/ Dan I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays", *Proceedings of the IEEE*, Vol.71, No.1, January 1983

/2/ S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, 1988

/3/ Sailesh K. Rao, *Regular Iterative Algorithms and Their Implementations on Processor Arrays*, Ph.D. 1986, Stanford University

/4/ Martin D. Meyer, Dharma P. Agrawal, "Adaptive Lattice Filter Implementations on Pipelined Multiprocessor Architectures", *IEEE Transactions on Communications*, Vol. 38, No. 1, January 1990

/5/ Michael K. Birbas, Dimitrios J. Soudris, Costas E. Goutis, "A New Method for Mapping Iterative Algorithms on Regular Arrays", *Communication, Control, and Signal Processing*, Elsevier Science Publishers B. V., 1990

/6/ Simon Haykin, *Adaptive Filter Theory*, Prentice-Hall, 1986

/7/ Walter J. Meyer, *Concepts of Mathematical Modelling*, McGraw-Hill Book Company, 1985

/8/ Frank S. Budnick, *Finite Mathematics with Applications*, McGraw-Hill Book Company, 1985

/9/ S. S. Rao, *Optimization - Theory and Applications*, Wiley Eastern Limited, 1984

Dr. Igor Ozimek
Institut Jožef Stefan, Jamova 39, Ljubljana
tel.: +386 1 477-3900
fax.: +386 1 251-9385, 426-2102
email: igor.ozimek@ijs.si

Prispelo (Arrived): 10.05.2002 *Sprejeto (Accepted): 28.06.2002*