Automatic Synthesis of Regular Architectures Optimized at the Bit Level

Florent de Dinechin, Patricia Le Moenner

This paper presents methods based on the formalism of affine recurrence equations for the synthesis of bit-level regular architectures from word-level (integer or real) algorithms. Because of bit-level dependency analysis, the arrays have optimal efficiency. We present two possible design flows leading to architectures based either on bit-parallel or bit-serial operators. The first one is fully automated.

1. INTRODUCTION

The formalism of Systems of Affine Recurrence Equations (SAREs) [5] is widely used for the design of regular parallel hardware for real-time signal-processing applications. It allows the stepwise refinement of an algorithm, from a very high level functional specification, down to a range of implementation-level descriptions suitable for simulation, compilation to sequential or parallel code, or hardware synthesis [10]. Alpha [8] is a strongly typed functional language based on the SARE formalism. The MmAlpha environment provides a set of automatic and semi-automatic tools to manipulate Alpha programs. These tools are formally proven to preserve the semantics of the specification.

One strong point of Alpha is the possibility of organizing SAREs into hierarchical modules [2]. So far these modules have been used at the top of the design flow (for the structured specification and analysis of algorithms), and at the bottom (to express hierarchical hardware designs). This paper shows how structuring may also be exploited in the synthesis design flow, for the automatic refinement of word-level algorithms into their bit-level equivalent.

An high-level algorithm is usually defined to operate on words of data which have abstract data-types, such as integer or real. The synthesis of such an algorithm, using the Alpha environment, gives an abstract systolic array also operating on words. At the implementation level, we have to convert these words to bit array representations, and replace operators on words, like $+$ or $\times$, with hardware components processing such bit arrays. So far, we had to use library (e.g. data-path) operators, which has the drawback of ignoring the possible bit-level parallelism. Our motivation was to change and improve this situation. For this purpose we present a new program transformation, to perform this bit-level refinement within the Alpha environment. Thus a word-level algorithmic description may undergo a bit-level dependency analysis and synthesis.

Depending on the step of the design-flow at which this bit-level refinement is made, a range of architectures may be obtained with various space/time complexity tradeoffs, using either bit-parallel or bit-serial operators [3]. In addition, this technique also keeps the design parameterized to a very late stage, another advantage on the library component approach. We demonstrate our ideas on the simple example of a real matrix-vector product in fixed point arithmetic, for which we give two possible architectures.

The organization of the paper is the following. In the next section, we present the Alpha language and environment. The third section details the synthesis leading to a bit-parallel design. The fourth section focuses on arrays using bit-serial operators synthesized in section 2.

2. THE ALPHA ENVIRONMENT

2.1. The Language

We introduce here the main features of the language with the help of Prog.1, a simple Alpha program which describes a classical binary adder. The interested reader is referred to [14, 2] for an extensive description of the language in its current version.

Alpha variables (here `A`, `B`, `S`, `X`, `C`) denote data arrays defined over a domain which is a con-

Program 1 Binary addition in ALPHA

```
1  system Plus: {W| W>1}
2            (A,B: {b| 0<=b<W} of boolean)
3    returns (S: {b| 0<=b<=W} of boolean);
4  var
5    X: {b| 0<=b<W} of boolean;
6    C: {b| 0<=b<=W} of boolean;
7  let
8    X[b] = A[b] xor B[b] xor Cin[b];
9    C[b] =
10        case
11          {| b=0} : 0[];
12         {| 1<=b} : A[b-1] and B[b-1]
13              or A[b-1] and C[b-1]
14              or B-1[b-1] and C[b-1];
15       esac;
16   S[b] =
17       case
18         {| b<W} : X[b];
19         {| b=W} : C[W];
20       esac;
21 tel;
```

Program 2 Binary multiplication in ALPHA

```
1  system Times: {W|W>1}
2            (A,B: {b| 0<=b<W} of boolean)
3    returns (X  : {b| 0<=b<W} of boolean);
4  var
5    P, Si: {b,m| 0<=b,m<W} of boolean;
6    So: {b,m| 0<=b<=W; 0<=m<W} of boolean;
7  let
8    P[b,m] = A[b] and B[m];
9    use {m| 0<=m<W} Plus[W] (Si,P)
10                   returns (So);
11   Si[b,m] =
12     case
13       {| m=0} : 0[];
14       {| m>0} : So[b+1,m-1];
15     esac;
16   X[b] = So[b+1,W-1];
17 tel;
```

vex polyhedron of some integer vector space $\mathbb{Z}^n$. Here the domain of `C` is `{b| 0<=b<W}`, where `W` is a size parameter defined in the header of the system: `{W| W>1}`. The values of a variable are defined on each point of this domain through recurrence equations involving the other variables, arithmetic or logical operators, and affine dependencies allowing to access the value of a variable at a different point (e.g. `C[b]` is defined as a function of `A[b-1]`, `B[b-1]` and `C[b-1]`). The `case` operator allows us to have several different expressions defining the values of a variable over distinct sub-domains (see the equation defining `C`, lines 9-15).

Program 2 describes the product of two binary-coded fixed-point reals. A real number $x \in [0 \dots 1]$ is coded as a bit string $b_{W-1} \dots b_1 b_0$ such that $x = \Sigma_{i=0}^{W-1} b_i . 2^{i-W}$. The product is performed as by hand (Fig.1), and only the most significant bits are kept. Each line of this figure is the product (logical *and*) of one bit of the second operand by all the bits of the first one, and these lines are accumulated thanks to "calls" to the system `Plus` of Prog.1. Note that the local variables (defined by the `var` keyword) are two-dimensional: the product involves a two-dimensional array of computations (see Fig.1).

This example introduces the `use` statement, which instantiates a regular collection of subsystems. Lines 9-10 should be read as follows:
Use a collection of instances of the system `Plus`*, indexed by* `{m| 0<=m<W}`*, the value of the size parameter being* `W` *for each instance. The inputs to the* `m`*-th instance are the* `m`*-th lines of arrays* `Si` *and* `P`*, and the output goes to the* `m`*-th column of array* `So`*.*

This description is an high-level SARE, in the sense that it doesn't contain any information about the order of execution of the instructions

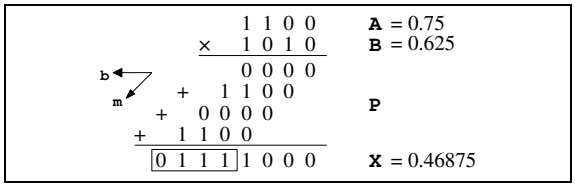

Figure 1. Product of two binary-coded fixed-point reals

(see equation defining P), and even less architectural information.

2.2. The tools

We now present MmAlpha, a Mathematica-based program transformation environment which allows us to derive implementation-level descriptions from high-level Alpha programs. We focus here on hardware synthesis, although imperative code [12] and data-parallel programs [11] may also be generated.

The reader is referred to [8] for a detailed description of the synthesis flow. A program like that of Prog.2 is first analyzed (type-checked) to ensure the consistency of the domains and expressions. Then it is *uniformized* [13, 15] to remove the data broadcasts and non-local communications. It is then *scheduled*, i.e. each computation of the program is assigned an affine time function consistent with the data dependencies [1]. Finally an affine *change of basis* is performed on the index space of each variable, so that one of the indices represents the *time* at which this variable is computed, and the other indices specify the *processor* on which the computation is performed, in some processor array whose shape is given by the resulting domains of the variables.

When this process is carried on Prog.2, we get an abstract description of the architecture described in Fig.2, whose core is partially given as Prog.3 (due to lack of space, some lines are deleted).

In this program, the data arrays are two-dimensional as in Fig.2, the index `t` represents the time and the index `m` is the processor index in the linear array of Fig.2. The declaration domains of the variables show that there are `W` processors, and that the `m`-th processor computes for `2m<=t<2m+W`. Line 8 shows how `A[t]` is input at time `t` on the first processor (`m=0`), and line 9 shows how it is then propagated from processor `m-1` to processor `m` through two registers (implied by the `t-2` dependency). The computation equations of lines 13-15 are interpreted as hardware operators, whereas the data translation equations are interpreted as registers. Finally the last equation shows that the `b`-th bit of the result is output by the last (`W-1`-th) processor at time `b+2W-1`. The complete program describes a virtual linear array (Fig.2) composed of elementary cells similar to the first $W-1$ cells of Lyon's multiplier [9].

Additional program transformations will lead to a program which may be interpreted as low-level hardware, expressed in a subset of Alpha called AlpHard[7] which has the following characteristics:

- hierarchical structuring (because design process is itself hierarchical);
- genericity (to allow for reuse of components);
- regularity.

AlpHard is organized in three levels: the lowest one is composed of two types of *purely temporal* systems, one describing elementary processors, and the second one expressing the initialization of the circuit's control signals. The next level specifies the *spatial combination* of components: for instance, the instantiation of a linear array of identical processors and the description of internal and external connections of the network. The highest level describes the *interface* – which is not always pure hardware – between the circuit and its environment. The transformation of an array expressed in Alpha into an AlpHard description involves several steps (control synthesis, space-time decomposition of the equations, hierarchical organization) which we won't present here due to lack of space.

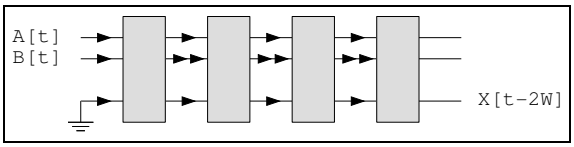

Figure 2. Bit-serial systolic array for the fixed-point multiplication

2.3. A matrix-vector product

The Alpha language also allows the user to express data arrays with `integer` or `real` abstract

Program 3 Fixed-point multiplier in ALPHA

```
1  system times ...
2  var
3    AA,BB,So,A_FA,B_FA,in_FA,S_FA,Cout_FA:
4      {t,m| 0<=m<W; 2m<=t<2m+W} of boolean;
5  let
6    AA[t,m] =
7        case
8          {| m=0} : A[t];
9          {| m>=1}: AA[t-2,m-1];
10       esac;
11   BB[t,m] = ... ;
12   A_FA[t,m] = ... ;
13   B_FA[t,m] = AA[t,m] and BB[t,m];
14   S_FA[t,m] = A_FA[t,m] xor B_FA[t,m]
15                         xor Cin_FA[t,m];
16   Cout_FA[t,m] = ... ;
17   So[t,m] =
18       case
19         {| t<=2m+W-1} : S_FA[t,m];
20         {| t=2m+W} : Cout_FA[t-1,m];
21       esac;
22   X[b] = So[b+2W-1,W-1];
23 tel
```

datatypes. The classical matrix-vector product is defined by the usual recurrence equation below:

$$\forall i \in \{1 \dots N\} \quad c_i = \sum_{j=1}^{N} a_{ij} b_j$$

This equation is implemented straightforwardly as the ALPHA SARE of Prog.4 where we have serialized the summation by accumulating the partial results in `C`.

The rest of this paper discusses the bit-level synthesis of such word-level specifications. The main point is to replace the abstract datatypes (`real` in our example) with their bit representation in order to synthesize real VLSI arrays. The simplest way to do this is to use library components (e.g. datapath operators). This method has several advantages (the simplicity of using of-the-shelf component, the availability of the latest VLSI technology) and one main drawback: we leave the SARE world, thus discarding the possible advantages of a bit-level dependency analysis.

Program 4 Matrix-vector product on abstract real data

```
1  system matvect :{N | 1<=N}
2           (M: {i,j| 1<=i,j<=N} of real;
3            V: {j| 1<=j<=N} of real)
4   returns (R: {i| 1<=i<=N} of real);
5  var
6    C : {i,j| 1<=i<=N; 0<=j<=N} of real;
7  let
8    C[i,j] =
9      case
10       {| j=0} : 0[];
11       {| j>0} : C[i,j-1] + M[i,j]*V[j];
12     esac;
13   R[i] = C[i,N];
14 tel;
```

We present two possible design flows: the first (sect.3) called the bit-parallel approach replaces, within the *specification* of the algorithm, the word-level operators with *specifications* of their bit-level implementation. For example the operator $\times$ is replaced in Prog.4 with the specification of the multiplier of Prog.2. This gives a specification which may then be scheduled and space/time transformed.

The second approach (sect.4) first independently schedules the specifications of the algorithm and the specification of the bit-level operators. This leads to a word-level array for the algorithm, and bit-level arrays for the operators. Then both are combined.

3. BIT-PARALLEL APPROACH

3.1. Simple Bit Extension

We introduce an automatic program transformation that replaces $+$ and $\times$ with instances of the corresponding ALPHA SAREs. For this purpose, one dimension is added to all the arrays of the real program, holding the bit array representation of each real. The bit size is parameterized.

For simplicity we present only the fixed-point bit representation of the reals. This work, however, may be straightforwardly adapted to integers (using unsigned, two's complement or Avize-

nis' redundant binary signed representation). We haven't considered floating point representations yet, but there is no theoretical obstacle apart from slightly reduced regularity.

This transformation is performed in four steps.

- First the operators are isolated by adding extra auxilliary variables, until they only appear in equations containing one and only one operator, without any dependency nor `case` statement, like the following:
  `A[i,j] = B[i,j]+C[i,j];`
  The transformations involved (e.g. to compute the domain of the extra variables) are well known and already used in the MMALPHA environment.

- Then a parameter, `W`, is added to the system, representing the word size, and all the variable declarations are augmented by one dimension (indexed by `{b| 0<=b<W}`) denoting the bit representation.

- Then all the expressions of the program are modified to take into account the extra dimension: a recursive search over the expression trees extends the dimensions of the `case` domains and affine dependencies. This again involves well-known SARE transformations.

- Finally, operator equations generated in the first step:
  `A[b,i,j] = B[b,i,j]+C[b,i,j]`
  are replaced with `use` statements like the following:
  `use` D_A `Plus[W] (B,C) returns (A);`
  where D_A is the declaration domain of `A` *before* the bit extension. This `use` line describes a collection of instances of system `Plus` corresponding to the collection of instances of additions expressed by the equation.

3.2. Bit-parallel Design Flow

Now we have automatically derived a specification of the bit-level array, which may undergo the synthesis process. All the systems (`matvect`, `Plus` and `Times` in our example) may be uniformized. Then the subsystems are *inlined* [2] and the resulting system is scheduled and space/time transformed, yielding a systolic array. The important point is that the scheduling takes into account the dependencies at the bit level. Thus the operators are pipelined maximally. Besides, we still have at this point a parameterized design where we may vary the bit size and the parameters of the problem (here `W` and `N`).

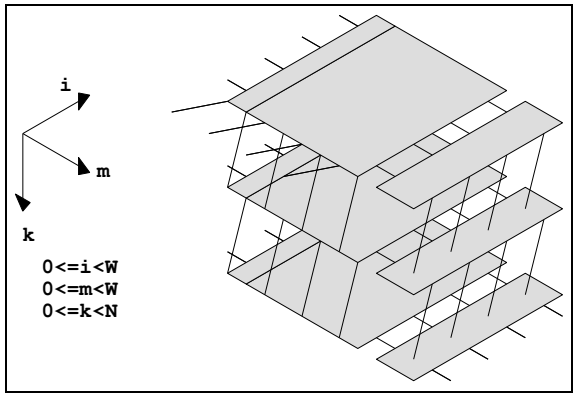

Figure 3. Three-dimensional virtual array for the fixed point matrix-vector product

In the case of the matrix-vector product, we get a 3-dimensional array (three processor indices, one time index) represented in Fig.3 (we do not give the ALPHA code due to lack of space). For synthesis, the three processor indices need projecting on a 2-dimensional space (the chip) as represented in Fig.4 and Fig.5. In these figures the arrows on the wires represent delays (registers). Note that some local data transfers in the 3-dimensional array lead to long (but still regular) wires in the actual circuit. The thick grey lines describe the bit arrays for each matrix or vector element, input and output in a *parallel skewed* manner. As previously, we read from the ALPHA program that, if the first bit of the matrix is input at time 0, the b-th bit of output vector element R_i is output from the `X` output of the last line of full adders at time:

$$T_R(b,i) = b + i + N + 2W + o(1)$$

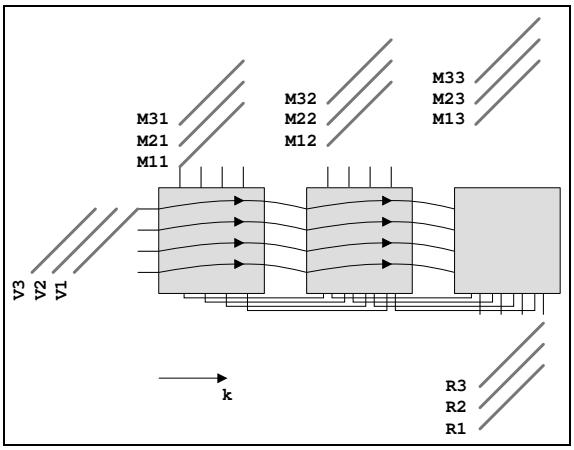

Figure 4. Bit-parallel systolic array

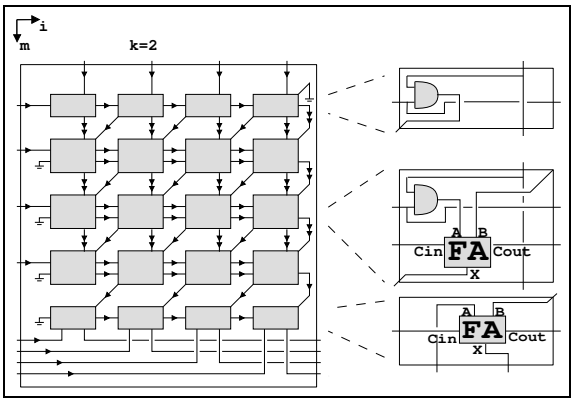

Figure 5. One cell of the bit-parallel array

One may check that, with only a little more control, this circuit may be used to perform several matrix-vector product in a pipelined manner.

The completion of the chain of transformations down to an AlpHard program is still under development.

4. BIT-SERIAL APPROACH

4.1. The Abstract Array

Serializing and scheduling the *real* matrix-vector product yields a SARE whose core is given as Prog.5, which represents a classic systolic array (Fig.6) similar to Kung and Leisersons' [6] but with unidirectional data-flow. In this figure, the registers are drawn as boxes. Program 5 shows that the output R_i is output from processor $i-1$ at time $T_R(i) = i + N - 2$. Note the time index `t` and the processor index `p`.

Program 5 Core of the scheduled Matrix-Vector Product

```
1  Vr[t,p] =
2    case
3      {| 0<=t<N; p=0}: V[t+1];
4      {| p<=t<p+N; 1<=p<N}: Vr[t-1,p-1];
5    esac;
6  Mr[t,p] = M[p+1,t-p+1];
7  C[t,p] =
8    case
9     {| t=p-1}: 0[];
10     {| p<=t}:  C[t-1,p] + Mr[t,p]*Vr[t,p];
11   esac;
12 R[i] = C[i+N-2,i-1];
```

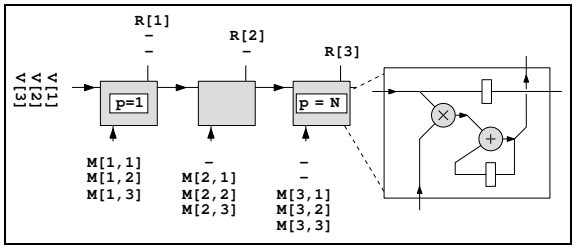

Figure 6. Real-typed systolic array for the matrix-vector product

4.2. Bit Extension in a Scheduled Design

The bit-extension can be done at this point or at the end of the transformation to AlpHard of the matrix-vector product. The idea is, as before, to replace the operators + and × with their bit-level counterpart. Basically, the bit extension is similar: we add one dimension to denote the bit arrays, indexed by a new index `b`. However, since

we now want to use the bit-serial operators obtained by scheduling the specifications of the bit-level operations, this new index has to be a time index which will be distinct from the time index `t` of the scheduled `matvect`. We will call `t` the *macroscopic* time index, denoting the number being processed, and `b` the *microscopic* time, denoting the bit index within one number. The actual time of an instruction will be a linear combination of `b` and the product `W`×`t`, and as such may not be expressed directly in the affine polyhedral SARE model (which only allows linear combinations of parameter and indices). Thus the bit-serial interpretation is inherently different from the usual systolic interpretation of an ALPHA program, for it involves the interpretation of several indices as a multidimensional time [4].

Therefore, the description of a bit-serial array in ALPHA is not trivial, and is not automated so far. First of all, the operators synthesized previously described *one* computation. We need to express that an operator will be reused for several computations, i.e. several (macroscopic) times. This is done with an `use` statement with a dimension extension on `t`, the macroscopic time. Then the word-level operators are replaced within the resulting SARE.

Figure 7 shows one cell of the bit-serial version of Fig.6. The b-th bit of the i-th vector element of the result is output from processor i at time:

$$T(b,i) = i \ + \ (N+1)W + b \ + \ o(1)$$

The latency of this implementation is obviously proportional to the product of the matrix size N and the bit size W, and therefore this design is linearly less efficient than the previous. Its silicon cost, however, is linearly better, for it has only two processor indices instead of three.

Remark that the figure shows that one of the word-level register of Fig.6 is synthesized as a single one-bit register, while the other is synthesized as `W` one-bit registers: depending on the data-dependencies of the program, we may interpret a macroscopic register (`(t->t-1)`) as one or `W` microscopic registers. Note that in a data-path implementation of the abstract `matvect`, the same dependency would always be synthesized as `W` registers in parallel.

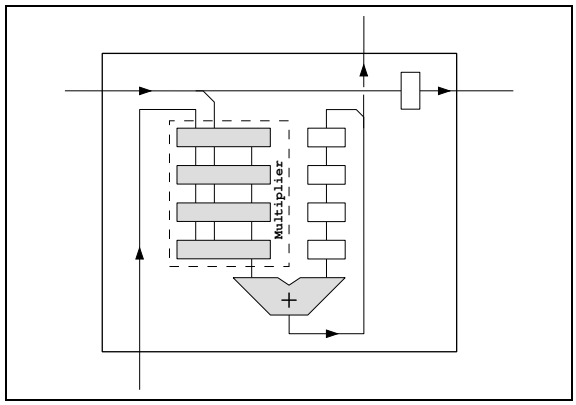

Figure 7. One cell of the bit-serial systolic array for the matrix-vector product

5. CONCLUSION & FUTURE WORK

We have introduced formal techniques allowing to synthesize, from word-level specifications, regular architectures optimized at the bit level where the word size is a generic parameter. Note that both arrays for the bit-level matrix-vector product have roughly the same space×time complexity.

The results presented in the second and third sections have been obtained in a fully automated manner by our tools. There is a potential for – at least partially– automating multidimensional time synthesis (section 4), and it is the subject of ongoing research. We are also investigating multidimensional scheduling techniques [4]. In addition to giving several space/time complexity trade-offs for each design, this will allow us to automatically synthesize SAREs for which our current tools can't find an affine schedule.

Other fields of application of similar techniques include:

- decomposing a computation of high dimensionality into any combination of sequential loops and regular hardware ;
- formal techniques for partitioning ;
- modeling multi-scale signal processing with SAREs.

REFERENCES

1. A. Darte and Y. Robert. – Scheduling uniform loop nests. – Technical Report 92-10, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, February 1992.
2. Florent de Dinechin, Patrice Quinton, and Tanguy Risset. – Structuration of the alpha language. – In W.K Giloi, S. Jahnichen, and B.D. Shriver, editors, *Massively Parallel Programming Models*, pages 18–24. IEEE Conmputer Society Press, August 1995.
3. Peter Denyer. – An introduction to bit-serial architectures for vlsi signal processing. – In B. Randell and P.C. Treleaven, editors, *VLSI Architectures*, pages 225–241. Prentice Hall International, 1983.
4. Paul Feautrier. – Some efficient solution to the affine scheduling problem, part ii, multidimensional time. – *International Journal of Parallel Programming*, 21(6), December 1992.
5. Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. – The organization of computations for uniform recurrence equations. – *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.
6. H.T. Kung and C.E. Leiserson. – Systolic arrays (for VLSI). – In *Sparse Matrix Proc. 1978*, pages 256–282. Society for Industrial and Applied Mathematics, 1978.
7. Patricia Le Moenner, Laurent Perraudeau, Sanjay Rajopadhye, Tanguy Risset, and Patrice Quinton. – Generating regular arithmetic circuits with alphard. – In *Massively Parallel Computing Systems (MPCS'96)*, May 1996.
8. Hervé Le Verge, Christophe Mauras, and Patrice Quinton. – The ALPHA language and its use for the design of systolic arrays. – *Journal of VLSI Signal Processing*, 3:173–182, 1991.
9. R. F. Lyon. – Two's complement pipeline multipliers. – *IEEE Trans. Comm.*, 24:418–425, April 1976.
10. D.I. Moldovan. – On the analysis and synthesis of VLSI algorithms. – *IEEE Transactions on Computers*, C-31(11), November 1982.
11. P. Quinton, S. Rajopadhye, and D. Wilde. – Derivation of data parallel code from a functional program. – In *IPPS*, Santa Barbara, USA, April 1995.
12. S. Rajopadhye and D. Wilde. – The naive execution of affine recurrence equations. – In *Application Specific Array Processors*, Strasbourg, France, July 1995.
13. S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. – On synthesizing systolic arrays from recurrence equations with linear dependencies. – In *Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503, New Delhi, India, Dec 1986. Springer Verlag, LNCS No 241.
14. Doran K. Wilde. – The alpha language. – Internal Report 827, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Januar 1994.
15. Yoav Yaacoby and Peter R. Cappello. – Converting affine recurrence equations to quasi-uniform recurrence equations. – In *AWOC 1988: Third International Workshop on Parallel Computation and VLSI Theory*. Springer Verlag, June 1988. – See also, UCSB Technical Report TRCS87-18, February 1988.