International Conference on Parallel Computing in Electrical Engineering (PARELEC'98)
Bialystok, Poland, September 2-5, 1998

Automatic Synthesis of a Parallel Architecture for Kalman Filtering using MMAlpha

Aurelien L. T. MOZIPO, Daniel MASSICOTTE, Patrice QUINTON*, and Tanguy RISSET*

Université du Québec à Trois-Rivières, Electrical Engineering Department
Research Group on Industrial Electronics
C.P. 500, Trois-Rivières, Québec, Canada, G9A 5H7, Tel.: +1-(819)-376-5071, Fax: +1-(819)-376-5219
E-mail: Aurelien_Landry_Mozipo_Tchoupou@uqtr.uquebec.ca, Daniel_Massicotte@uqtr.uquebec.ca
*INRIA Rennes, University of Rennes 1 and IRISA of Rennes
Parallel VLSI Architectures Team
Campus de Beaulieu, 35042 Rennes Cedex, Rennes France, Tel.: +33-2.99.84.71.85, Fax: +33-2.99.84.71.00
E-mail: Patrice.Quinton@irisa.fr, Tanguy.Risset@irisa.fr

Abstract - **The intensive computations involved in the Kalman filtering are not feasible for many practical applications. Despite the great versatility of this method, this phenomenon largely reduces the use of Kalman filtering in real time applications and in all other fields where throughput is an important criterion. In this paper, we propose the use the MMAlpha tool as an innovative technique which gives automatically a parallel architecture of the covariance Kalman filter. Alpha is a functional language developed for the synthesis of regular architectures from recurrence equations. Many other Kalman-filter-based parallel architectures have been proposed, but, unlike ours, they are not derived automatically by means of a tool. In addition to having the advantage of being fast, this method leads to a systolic architecture of an array of M^2+1 elementary processors and a timesteps per iteration of 7M+6, where M is the dimension of the covariance matrix. A comparative study is done with other architectures proposed in the literature. As an application, we use Kalman filtering for signal reconstruction and specifically for adaptive channel equalization.**

I. INTRODUCTION

With the emergence of real time applications with intensive demanding computations, the design of dedicated processors has been considerably developed, either in the methodology or in means to accelerate the computation rate within the processor. Modern VLSI design techniques are numerous and all have the same objectives: to minimize the number of timesteps between updates and to maximize the number of operations carried out during one clock cycle. Early techniques developed aimed at executing operations as quickly as possible, but with the emergence of the parallel computing techniques in VLSI circuits like systolic arrays [15], a new approach consisting of performing many operations during the same cycle quickly spread among the integrated circuits designers. Consequently, several authors used this approach to develop parallel architectures based on information and covariance Kalman filter in various application fields (e.g. [2]-[4]). These architectures are derived by linear algebraic transformations like Faddeev algorithm and Givens rotations (e.g. [2], [3]), or manually by computing in advance the Kalman gains before feeding them into the process [16], [19]. Obviously, this last method does not integrate the resolution of the Riccatti equation, which however constitutes the biggest obstacle to the integration of the Kalman filter on a silicon chip. Many other Kalman filter based parallel architectures have been proposed (e.g. [5]-[10]).

We propose the application of a set of tools that give automatically a parallel architecture of a Kalman filter, based on the formalism of systems of affine recurrence equations [11]-[13]. The tools are integrated within the environment called MMAlpha and functional in Mathematica®. These advanced techniques for parallel computing in very large-scale integration circuits were proposed in [11-13], [18], [20], [21]. In the final architecture derived with these tools, parallelism is maximal and therefore, the time between two consecutive output samples is minimized.

In this work, the MMAlpha environment is applied to the Kalman filter to obtain a parallel architecture. Since the Kalman filter is a linear optimal estimator, it can also be used for state reconstruction of a time varying dynamic system. In this paper we apply the Kalman filter to solve the ill-posed problem in signal reconstruction and specifically for adaptive equalization [14].

In section II, we discuss the covariance Kalman filtering and its application to adaptive channel equalization. An extended summary of the MMAlpha environment is given in section III, and the application of this tool to the Kalman filter to obtain a systolic architecture is described in section IV. Section V presents the systolic architecture obtained. Finally, in section VI, we compare the performance of our design with some previously published architectures.

This work was supported by Natural Sciences and Engineering Research Council of Canada, Fonds FCAR Québec, and Association of Universities and Colleges of Canada.

II. THE KALMAN FILTERING AND SIGNAL RECONSTRUCTION

The implementation in a VLSI technology of Kalman filter based algorithms for specific applications is required in various fields such as signal processing, communications, as well as control systems. This interest is explained by the fact that the Kalman filtering is a very powerful tool in real time estimation process [1]. This filter is based on the principle of estimating the state of a system based on noisy measurements in a stochastic environment, by minimizing the estimated mean squared error. Consider a discrete time-varying dynamic system defined by the following state equation:

$$\mathbf{x}_{k+1} = \mathbf{\Phi}_k \mathbf{x}_k + \mathbf{B}_k \mathbf{u}_k + \mathbf{w}_k, \quad \mathbf{x}_0 = 0 \tag{1}$$

$$\tilde{\mathbf{y}}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k \tag{2}$$

for k=1,2,… and where $\mathbf{x}_k$ is the (M×1) state vector, $\tilde{\mathbf{y}}_k$ is the (1xN) measurement vector and **u**(k) is the (P×1) control vector. $\mathbf{\Phi}_k$, $\mathbf{B}_k$ and $\mathbf{H}_k$ are known matrices. Also, **v** and **w** are two sequences of non-correlated white noise with known covariance matrix $\mathbf{R}_k^v$ and $\mathbf{R}_k^w$ respectively. The conventional Kalman filtering equations can be summarized in the following covariance form [2]:

$$\hat{\mathbf{x}}_{k+1/k} = \mathbf{\Phi}_k \hat{\mathbf{x}}_{k/k} + \mathbf{B}_k \mathbf{u}_k \tag{3}$$

$$\mathbf{P}_{k+1/k} = \mathbf{\Phi}_k \mathbf{P}_{k/k} \mathbf{\Phi}_k^T + \mathbf{R}_k^w, \quad \mathbf{P}_{0/0} = \mathbf{I} \tag{4}$$

$$\mathbf{V}_{k+1} = \mathbf{H}_k \mathbf{P}_{k+1/k} \mathbf{H}_k^T + \mathbf{R}_k^v \tag{5}$$

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1/k} \mathbf{H}_{k+1}^T \mathbf{V}_{k+1}^{-1} \tag{6}$$

$$\hat{\mathbf{x}}_{k+1/k+1} = \hat{\mathbf{x}}_{k+1/k} + \mathbf{K}_{k+1}\left[\tilde{\mathbf{y}}_{k+1} - \mathbf{H}_{k+1}\hat{\mathbf{x}}_{k+1/k}\right] \tag{7}$$

$$\mathbf{P}_{k+1/k+1} = \mathbf{P}_{k+1/k} - \mathbf{K}_{k+1}\mathbf{H}_{k+1}\mathbf{P}_{k+1/k} \tag{8}$$

where $\hat{\mathbf{x}}_{k+1/k}$ is the prediction of state **x** at time k+1 given measurements and information up to and including time k; $\hat{\mathbf{x}}_{k+1/k+1}$ is the estimation of state **x** at time k+1 given measurements and information up to and including time k+1; $\mathbf{P}_{k+1/k+1}$ is the covariance of estimation error; $\mathbf{P}_{k+1/k}$ is the covariance of prediction error; $\mathbf{K}_{k+1}$ is the Kalman gain.

We apply the Kalman filter to adaptive channel equalization problem. This problem is defined by the convolution operation of a signal a_k crossing a channel (e.g. conversion system in a measurement system, modem, satellite communication, cellular phone), represented by a impulse response function vector $\mathbf{h}_k$, where the output $\tilde{y}_k$ is a scalar corrupted with additive noise v_k. So, the dimensions of the matrices and vectors in the equations (1)-(8) are N=1 and P=1. The discrete form of the convolution equation is:

$$\tilde{y}_k = \sum_{m=1}^{M} h_{k\text{-}m} a_m + v_k \quad \text{for } k = 1,2,3,... \tag{9}$$

The signal which is applied to the channel, defined as the measurand signal, can be estimated by numerical methods on the basis of an a priori knowledge of the impulse response of the conversion system and the measured samples of the output. This operation is a numerically ill-conditioned rule. The solutions given by LMS and RLS algorithms have been proposed in [14] and by Kalman filter in [17], [22]. Here, we used the model gives by the equations (1) and (2) without the control input $\mathbf{u}_k$ and we considered the matrix $\mathbf{\Phi}$ as invariant. By normalizing the covariance matrices, we obtain the following form for equations (3)-(8) respectively [22]:

$$\hat{\mathbf{x}}_{k+1/k} = \mathbf{\Phi}\hat{\mathbf{x}}_{k/k} \tag{10}$$

$$\mathbf{P}_{k+1/k} = \mathbf{\Phi}\mathbf{P}_{k/k}\mathbf{\Phi}^T + \mathbf{b}\beta\mathbf{b}^T, \quad \mathbf{P}_{0/0} = \mathbf{I} \tag{11}$$

$$\mathbf{V}_{k+1} = \mathbf{h}_k \mathbf{P}_{k+1/k} \mathbf{h}_k^T + 1 \tag{12}$$

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1/k} \mathbf{h}_{k+1}^T \mathbf{V}_{k+1}^{-1} \tag{13}$$

$$\hat{\mathbf{x}}_{k+1/k+1} = \hat{\mathbf{x}}_{k+1/k} + \mathbf{K}_{k+1}\left[\tilde{\mathbf{y}}_{k+1} - \mathbf{h}_{k+1}\hat{\mathbf{x}}_{k+1/k}\right] \tag{14}$$

$$\mathbf{P}_{k+1/k+1} = \mathbf{P}_{k+1/k} - \mathbf{K}_{k+1}\mathbf{h}_{k+1}\mathbf{P}_{k+1/k} \tag{15}$$

Moreover, the measurement noise is taken stronger than the value in [14] to appreciate the accuracy of data correction with the Kalman Filter. Our experiment has shown that the Kalman filter gives better results than the LMS and RLS algorithms.

The next step of the work is to translate equations (10)-(15) in the Alpha language and then to derive a parallel architecture with MMAlpha tool.

III. SUMMARY OF THE MMALPHA ENVIRONMENT

The tool we use to derive these parallel architectures is the Alpha language and its development environment called MMAlpha. Alpha is a functional language developed for the synthesis of regular architectures from recurrence equations [12]. In Alpha, an algorithm is described as a set of equations on variables defined on multi-dimensional domains. Each variable or expression of the language is actually a function from a set of integer coordinate points satisfying linear inequalities, to a set of values. The synthesis process consists of applying a sequence of semantic preserving transformations which map the initial specification of the algorithm to an architecture which supports its execution. The final description can be translated into VHDL in order to generate a VLSI architecture.

All tools needed to perform these transformations are implemented in the MMAlpha environment as a set of Mathematica® packages together with C libraries.

The design process starts from an algorithmic level description of the application which can be readily obtained from the equations of the process such as those of Part II. This description can be organized as a hierarchy where basic linear algebra algorithms such as matrix vector multiplication, matrix multiplication, etc. are first described and then used in the application. A C program which evaluates this description can be automatically generated in order to check the correctness of the initial specification by simulation. Then the initial description undergoes a series of transformations which deliver an abstract architecture. Among these transformations, localization and scheduling are the most important ones. Localization (also called uniformization or pipelining in the literature) replaces non-local calculations by local ones. Scheduling orders the calculations in such a way that the evaluation of a given variable can be performed after that of its components. Scheduling amounts to solving an integer linear programming problem whose unknowns are the coefficients of the affine function which defines the time at which each variable may be evaluated [20]. For example, a variable V[i,j] is scheduled at time ai+bj+c, and the coefficients a, b, c are unknowns. Scheduling also provides interesting information on the total time needed to execute the algorithm. Once a schedule has been found, a change of basis allows all calculations to be expressed in a new index space giving the evaluation time and the processor number where a calculation is to be performed.

From this abstract architecture, the design of an actual architecture consists of applying a sequence of low-level transformations which brings the description to a net-list format called AlpHard. The transformation process is almost automated, the MMAlpha environment behaving as a compiler which automatically maps one description level to the next one. Eventually, one obtains a VHDL model to implement in a VLSI technology (e.g. FPGA, CMOS) [13].

IV. TRANSFORMATION PROCESS

The design methodology within the MMAlpha environment explained above is applied to equations (10) to (15) to obtain a scheduling for all output and internal variables and a netlist of a processors array which defines the architecture. Equations (10) to (15) are rewritten into the Alpha language and are analyze to remove syntax error and check the domains of the equations by static analysis. Then we inline all subsystems [18]: this transformation flattens all structured Alpha expressions such that the resulting program can be translated in C with the *WriteC* translator. The C program obtained is executed and results are compared to those obtained with Matlab®.

After these preliminary transformations, the effective architecture derivation can begin. First, we pipeline some broadcasted variables along given directions. Computations involved in the pipeline process are matrix-matrix, matrix-vector and vector-vector multiplications. After the pipelining, we can look for a scheduling for the Alpha program. The goal of the *scheduler* is to find a valid execution order with respect to a particular criterion. The time is considered as a discrete single clock. The overall idea of the scheduling process is to build a linear programming problem and solve it with a software tool [18].

The scheduling for the computation of all variables of the Alpha program is summarized in Table I. Table I shows the order in which each variable is computed, the total time needed to compute it, and the number of timesteps. The number of timesteps is defined both without pipelined and with pipelined on the scheduling. All these information are provided automatically by the scheduler of MMAlpha. This schedule is made explicit by applying a time space reindexing to the Alpha program. Then the control signals are generated and we derive the AlpHard model of the architecture.

Table I : Timesteps derivation from the scheduling given by MMAlpha.

Eq.	Variable	Operation	Scheduling time given by MMAlpha i=1,2,…,M m=1,2,…,M	Number of timesteps	
				without pipeline	**with pipeline**
(10)	$\hat{\mathbf{x}}^{\text{int}}$	$\mathbf{\Phi}\hat{\mathbf{x}}_{k/k}$	1+m+M	2M-1	M
(14)	$\hat{y}$	$\hat{\mathbf{x}}^{\text{int}}\mathbf{h}_{k+1}$	7+2M	M	M
	δ_y	$\tilde{\mathbf{y}}_{k+1} - \hat{\mathbf{y}}$	8+2M	0	
(11)	$\mathbf{A}_1$	$\mathbf{P}_{k/k}\,\mathbf{\Phi}^T$	2+i+m+M	3M-1	
	$\mathbf{P}_{k+1/k}$	$\mathbf{\Phi}\,\mathbf{A}_1 + \mathbf{b}\beta\mathbf{b}^T$	1+i+m+2M	3M-1	M
(13) (12)	$\mathbf{A}_2$	$\mathbf{P}_{k+1/k}\mathbf{h}_k$	1+m+3M	2M-1	M
	$\mathbf{A}_3$	$\mathbf{P}_{k+1/k}\mathbf{h}_{k+1}$	2+m+4M		
(12)	$\mathbf{V}^{\text{int}}$	$\mathbf{h}_k\mathbf{A}_2$	1+4M	M	M+1
(13)	$\mathbf{V}_{k+1}^{-1}$	$1/(\mathbf{V}^{\text{int}}+1)$	3+4M	1	1
(13)	$\mathbf{K}_{k+1}$	$\mathbf{A}_3\mathbf{V}_{k+1}^{-1}$	3+m+4M	2	2
(14)	$\hat{\mathbf{x}}_{k+1/k+1}$	$\hat{\mathbf{x}}^{\text{int}} + \delta_y\mathbf{K}_{k+1}$	5+m+4M	2	2
(15)	$\mathbf{P}^{\text{int}}$	$\mathbf{I} - \mathbf{K}_{k+1}\mathbf{h}_{k+1}$	4+i+m+4M	M	M
	$\mathbf{P}_{k+1/k+1}$	$\mathbf{P}^{\text{int}}\mathbf{P}_{k+1/k}$	3+i+m+5M	3M-1	M
Total				16M	7M+6

V. ARCHITECTURE

We show here the architecture that one obtains after applying the transformations of MMAlpha. This architecture is made of a M×M processing elements (PE) square array; each of which is a multiplier-accumulator (MAC), with a control input which defines the operating mode, i.e. it selects appropriate input data to feed into the arithmetic unit, and the data to be output. One divider is needed to compute $1/(\mathbf{V}^{\text{int}}+1)$. Therefore, the overall architecture has M^2+1 PEs arranged as shown later in this section. One advantage presented by this architecture is that many intermediate variables and data are used immediately after their computations to pipeline the scheduling. Hence, they stay in the array for the next timestep. However, we need to store the constant data $\mathbf{b}\beta\mathbf{b}^T$ and $\mathbf{\Phi}$, and some variables $\hat{\mathbf{x}}_k$, $\mathbf{h}_k$, $\mathbf{h}_{k+1}$, $\mathbf{P}_k$, $\mathbf{P}_{k/k+1}$, $\mathbf{V}_{k+1}^{-1}$ and δ_y. Figure 3 shows two storage units; each can be constituted of M memories. $\mathbf{P}_k$ and $\mathbf{P}_{k/k+1}$ matrices are symmetric, we need to store only their lower (or upper) triangular part respectively.

We explain how the architecture works by showing in figure 1, as an example, the computation of equation (11). In this figure, the control signals and other I/O pads are omitted for the sake of simplicity. Here, each processing cell uses two operating modes when computing equation (11). The first operation is $\mathbf{P}_{k/k}\mathbf{\Phi}^T = \mathbf{A}_1$ in mode 1, the result $\mathbf{A}_1$ is stored in the array, then $\mathbf{\Phi}\mathbf{A}_1$ is computed in mode 2 and the result

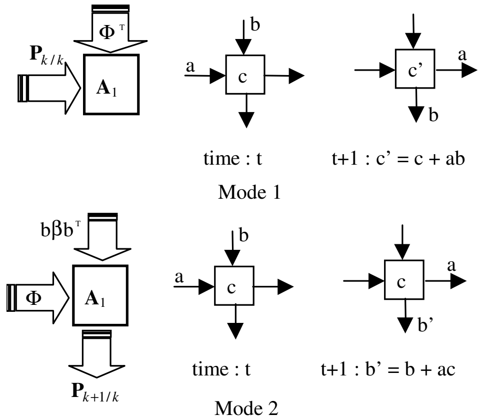

Figure 1: Data flow and MAC cell for operating mode 1 and 2.

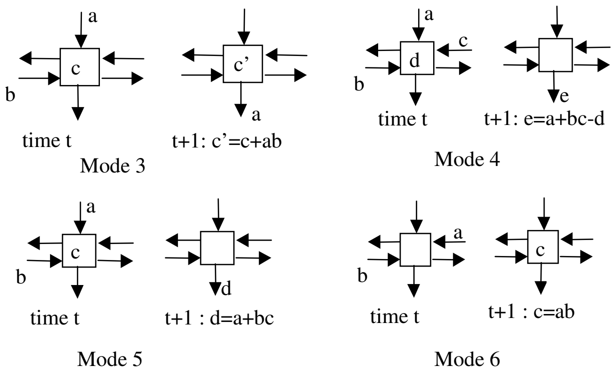

Figure 2: Operating modes

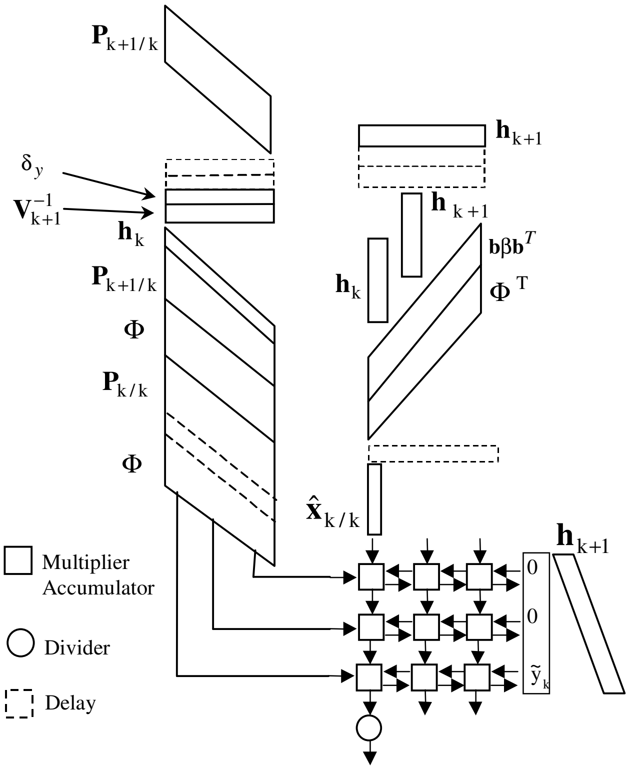

Figure 3: Overall architecture and data arrangements for one update.

is added to $\mathbf{b}\beta\mathbf{b}^T$. This computation involves two operating modes for all the M^2 PEs of the array. Each PE has several operating modes determined by variables scheduled to be computed within it, and also according to the direction (left, right, top and bottom) throw which these variables are pipelined into the PE.

The overall systolic architecture is given in figure 3 for M=3. The operation of the array processors is realized in 10 steps as follows:

Step 1: $\mathbf{\Phi}$ and $\hat{\mathbf{x}}_{k/k}$ are fetched into the array, $\tilde{y}_k$ is loaded to PE(M,M), $\hat{\mathbf{x}}^{\text{int}}$ is then computed in the first column, with the PEs operating in mode 3. After M timesteps $\hat{x}_{k/k}(1)$ is available in PE(1,1) and it is stored in the internal register for the next step; then it moves step by step toward the last PE of the row, PE(1,M). And so do all other elements of the vector immediately after they are computed.

Step 2: $\hat{\mathbf{x}}^{\text{int}}$ computed in step 1 circulates in the array and meets $\mathbf{h}_{k+1}$ in the last column. $\hat{\mathbf{x}}^{\text{int}}$ and $\mathbf{h}_{k+1}$ are multiplied element by element. The scalar δ_y is obtained by successive downward accumulations of the obtained elements, as shown in figure 2 in mode 4. Note that d= $\tilde{y}_k$ for PE(M,M) and d=0 in PE(i,M) for all i=1,..,M-1, at the beginning. A unit delay is observed before the next data input, to allow the multiplication of $\hat{\mathbf{x}}^{\text{int}}$ by $\mathbf{h}_{k+1}$ to take place. Then $\mathbf{P}_{k/k}$ and $\mathbf{\Phi}^T$ are fetched into the array. $\mathbf{A}_1$ is computed in all the PEs of the array, with operating mode 1; the result stays in the array.

Step3: $\mathbf{\Phi}$ and $\mathbf{b}\beta\mathbf{b}^T$ are fetched into the array. $\mathbf{P}_{k+1/k}$ is computed with operating mode 2 shown above. The results leave the array by the bottom.

Step 4: $\mathbf{P}_{k+1/k}$ is fetched in from the left, $\mathbf{h}_k$ and $\mathbf{h}_{k+1}$ are fetched in row 1 and 2 respectively, with a unit delay between them. $\mathbf{A}_2$ and $\mathbf{A}_3$ are computed by PEs of the first two rows operating in mode 1. Results are stored in corresponding rows of the array.

Step 5: $\mathbf{h}_k$ is fetched in from the left of the array after $\mathbf{P}_{k+1/k}$. It is multiplied element by element in the first row, then accumulated downward to give $\mathbf{V}^{\text{int}}$ (operating mode 5).

Step 6: The result of step 5 is sent to the divider to compute $\mathbf{V}_{k+1}^{-1}$.

Step 7: $\mathbf{V}_{k+1}^{-1}$ is then fetched from the left in all rows of the first column. The content of the second column is fed back to the first column for an element by element multiplication (operating mode 6). The result, $\mathbf{K}_{k+1}$, is stored in the array.

Step 8: δ_y is fetched from the left to all the rows of the first column. $\hat{\mathbf{x}}^{\text{int}}$ is retrieved from the internal register and $\hat{\mathbf{x}}_{k+1/k+1}$ is computed. The result $\hat{\mathbf{x}}_{k+1/k+1}$ is output and $\mathbf{K}_{k+1}$ is stored in the internal register of the first column.

Table II: Comparison to other covariance Kalman filter based designs

Architecture	Number of PE	Number of Timesteps per iteration	PE Utilization	
			M=3	M>>1
Irwin [2]	M(3M+1)/2	9M+9	41%	44%
Yeh [3] Scheme A	$4M^2$	16M	13%	9%
Yeh [3] Scheme B	$8M^2$	8M	13%	9%
Proposed	M^2+1	7M+6	82%	86%

Step 9: $\mathbf{K}_{k+1}$ is retrieved from the internal register, h is fetched in from the top of the array. $\mathbf{K}_{k+1}$ circulates from left to right and meets $\mathbf{h}_{k+1}$ which circulates from top to bottom for element by element multiplication. The obtained matrix is subtracted from identity matrix **I**. The result $\mathbf{P}^{\text{int}}$ is stored in the array.

Step 10: $\mathbf{P}_{k+1/k}$ is fetched in from the left, and $\mathrm{P}_{k+1/k+1}$ is computed. The result moves out by the bottom. All PEs operate in mode 2.

VI. Comparison of Performances

Table II shows a comparative study with some architectures based on the Kalman filter defined by the Eq. (10)-(15) such as the trapezoidal array designed by Irwin [2] and both architectures scheme proposed by Yeh [3]. The comparison is based on number of PEs, the computation time defined by the number of cycles to execute the computation applied to one sample $\tilde{y}_k$, and the PE utilization. The comparison was based on the assumption that the clock frequency is the same for all architectures, that the complexity of PEs is equivalent, and that all data needed to execute all operations are available.

The PE utilization introduced here is computed as the ratio between the average number of active processors and the number of processors. The average number of active processors is defined as the total number of arithmetic operations divided by the number of timesteps per iteration or per sample k [8]:

$$\text{PE Utilization} = \frac{\text{nb of arithmetic operations}}{\text{nb timestep} \times \text{nb PE}} \tag{16}$$

From the Table I we obtain $6M^3+5M^2+5M+1$ arithmetic operations. Table II shows that our architecture presents the smallest computation time, the smallest number of PEs, and the highest processor utilization.

VII. Conclusion

In this paper we have applied the MMAlpha environment to obtain automatically a systolic architecture for the covariance Kalman filter. This architecture achieves a processor utilization of more than 82% with a speed of 7M+6 timesteps per iteration. These performances are due to the fact that we have used a particular software tool, the MMAlpha environment, to solve the linear programming problem that appear among variables. This leads us therefore to an architecture in which the parallelism is maximal. The algorithm used has been applied successfully to solve the adaptive channel equalization problem. Applications of these systolic architectures are possible in a wide variety of fields such as control, telecommunications, metrology, biomedical engineering, seismology and spectrometry. The next step of this work will be to implement the proposed architecture in a VLSI technology (0.5µm CMOS or FPGA).

References

[1] H. W. Sorenson, "Kalman Filtering: Theory and Application", IEEE Press, 1985.
[2] G. W. Irwin, "Architectures for Control", Chap. 9 of Algorithms and parallel VLSI architectures, Elsevier Science, 1991, pp. 431-443.
[3] H. G. Yeh, "Systolic Implementation on Kalman Filters", IEEE Trans. on acoustics, speech, and signal processing, Vol.36, No 9, pp.1514-1517, 1988.
[4] R. A. Lincoln and K. Yao, "Efficient Systolic Kalman Filtering Design by Dependence Graph Mapping", VLSI Signal Processing III, Edited by R. W. Brodersen et H.S. Moscovitz, IEEE Press, 1988, pp.396-407.
[5] F. M. F. Gaston and G. W. Irwin, "Systolic approach to square root information Kalman filtering", International Journal of Control, vol. 50, no. 1, 1989, pp. 225-248.
[6] F. Gaston and G. Irwin, "VLSI architectures for square root covariance Kalman filtering", Proc. SPIE, vol.1152, 1989, pp. 44-55.
[7] M. R. Azimi-Sadjadi, T. Lu, and E. M. Nebot, "Parallel and Sequential Block Kalman Filtering and Their Implementation Using Systolic Arrays", IEEE - Transactions on Signal Processing, vol. 39, N° 1, January 1991, pp. 137-147.
[8] S. Y. Kung and J. N. Hwang, "Systolic Array Designs for Kalman Filtering", IEEE - Transactions in Signal Processing, vol. 39, N° 1, January 1991, pp. 171-182.
[9] G.M. Megson, "Fast Multi-layer Systolic Arrays for Kalman Filtering", Algorithms and Parallel VLSI Architectures, Vol.B: Proceedings, E.F. Depretter and A. Deprettere and A.-J. van der Veen (eds.), Elsevier Science Publishers B.V., 1991, Chap. 15, pp. 145-154.
[10] P. Rao and M. Bayoumi, "An Algorithm Specific VLSI Parallel Architecture for Kalman Filter", IEEE Press: VLSI Signal Processing, IV, 1991, pp. 264-273.
[11] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays", Journal of VLSI Signal Processing, Vol. 1, No 2, October 1989, pp. 95-113.
[12] H. Le Verge, C. Mauras, and P. Quinton, "The ALPHA language and its use for the design of systolic arrays", Journal of VLSI Signal Processing, Vol.3, 1991, pp. 173-182.
[13] P. Le Moenner et al., "Generating Regular Arithmetic Circuits with ALPHARD", MPCS'96, Ischia, Italy, 6-9 May 1996.
[14] S. Haykin, "Adaptive Filter Theory", Prentice Hall, 1996, Chap. 9.
[15] H. T. Kung, "Why systolic architectures", IEEE Computer, Vol. 15, pp. 37-46, Jan. 1982
[16] D. Massicotte, "A Systolic VLSI Implementation of Kalman-Filter-Based Algorithms for Signal Reconstruction", IEEE Int. Conf. Acoustics, Speech, and Signal Processing, Seattle, 12-15 May 1998, pp. 3029-3032.
[17] D. Alba, G. R. Meira, "inverse optimal filtering method for the instrumental spreading correction in size chromatography", J. of Liquid Chromatography, 7(14), 2833-2862 (1984), by Marcel dekker, Inc.

[18] Getting Started with Alpha, API-COSI, Internal report, IRISA, Rennes, Sept. 97.
[19] S. Kiaei, U. B. Desai, "Independent Data Flow Wavefront Array Processors for Recursive Equations", in proc. VLSI signal processing II, IEEE press, NY, 1986, pp. 152-164.
[20] S. Balev, P. Quinton, S. Rajopadhye, and T. Risset, "Linear Programming Models for Scheduling Systems of Affine Recurrence Equations - a Comparative Study", SPAA?98, Puerto Vallarta, Mexico, pp 250-258, June 1998.
[21] "A langage for synthesis of regular architectures", http://www.irisa.fr/api/ALPHA/welcome.html, June 1998.
[22] D. Massicotte, R. Z. Morawski, and A. Barwicz, "Incorporation of a Positivity Constraint Into a Kalman-Filter-Based Algorithm for Correction of Spectrometric Data", IEEE Trans. Instr. and Meas., Vol. 44, No 1, February 1995, pp. 2-7.

APPENDIX

Alpha program describing one step of Kalman filtering

The following is the Alpha program for one step of the Kalman filter. This program contains the definition of four subsystems. System OneStep is the main program, and corresponds, up to the renaming of some variables, to the equations given in Table I. This system calls three other subsystems, called matmult (for matrix multiplication), matvect (matrix vector multiplication) and dotprod (for dot product). Each subsystem contains a list of input variables, a list of output variables (preceded by the keyword returns), a list of local variables, and a list of equations. In system OneStep, most of the equation are so-called use statements, and behave (roughly speaking) much as subroutine calls in a conventional language. Notice that all systems are parameterized by the size parameter M.

```
-- OneStep of Kalman
system OneStep :{M | 2<=M}
        (yb :  real;
         Ip : {m,i | 1<=m<=M; 1<=i<=M} of real;
         bbt : {m,i | 1<=m<=M; 1<=i<=M} of real;
         phi : {m,i | 1<=m<=M; 1<=i<=M} of real;
         phit : {m,i | 1<=m<=M; 1<=i<=M} of real;
         H : {m | 1<=m<=M} of real;
         Hz : {m | 1<=m<=M} of real;
         xchapz : {m | 1<=m<=M} of real;
 P : {m,i | 1<=m<=M; 1<=i<=M} of real)
   returns (K : {m | 1<=m<=M} of real;
         xchap : {m | 1<=m<=M} of real;
         Pkk : {m,i | 1<=m<=M; 1<=i<=M} of real);
var
 xint : {m | 1<=m<=M} of real;
 Ychap :  real;
 I :  real;
 Veint :  real;
 invVe :  real;
 V2 : {m | 1<=m<=M} of real;
 V3 : {m | 1<=m<=M} of real;
 PP, PP1, V1 : {m,i | 1<=m<=M; 1<=i<=M} of real;
 Pint : {m,i | 1<=m<=M; 1<=i<=M} of real;

let
 -- Equation 9
 use  matvect[M] (phi, xchapz) returns  (xint) ;
 -- Equation 7
 use  dotprod[M] ( xint, H) returns  (Ychap) ;
 I[] = yb[] - Ychap[];
 -- Equation 10
 use  matmult[M] (P, phit) returns (V1);
 use  matmult[M] (phi, V1) returns (PP1);
 PP = PP1 + bbt;
 -- Equation 11
 use  matvect[M] (PP, H) returns  (V2) ;
 use  dotprod[M] (Hz, V2) returns  (Veint) ;
 -- Equation 6
 use  matvect[M] (PP, Hz) returns  (V3) ;
 invVe[] = 1 / (Veint[]+1[]);
 K[m] = V3[m] * invVe[];
 -- Equation 7
 xchap[m] = xint[m] + K[m] * I[];
 -- Equation 8
 Pint[m,i] = Ip[m,i] - K[m] * H[i];
 use  matmult[M] (Pint, PP) returns  (Pkk) ;
tel;
```

```
-- Matrix matrix multiplication
-- Inputs: a, b: square matrices of size M
-- Outputs: c: square matrix of size M
system matmult : {M |M>1}
 (a,b : {i,j | 1<=i,j<=M} of real)
returns
 (c : {i,j | 1<=i,j<=M } of real);
var
 C : {i,j,k | 1<=i,j<=M; 0<=k<=M} of real;
let
 c[i,j] = C[i,j,M];
 C[i,j,k] = case
      {|k=0} : 0[];
      {|1<=k<=M} : C[i,j,k-1]+a[i,k]*b[k,j];
 esac;
tel;

-- Matrix vector multiplication
-- Input: a: a square matrix of size M
--       v: a vector of size M
-- Output: c: a vector of size M
system matvect : {M | M>1}
          (a : {i,j | 1<=i,j<=M} of real;
           v : {i | 1<=i<=M} of real)
      returns  (c : {i | 1<=i<=M} of real);
var
      C : {i,j | 1<=i<=M; 0<=j<=M} of real;
let
      C[i,j] = case
          {| j=0} : 0[];
          {| j>=1} : C[i,j-1] + a[i,j]*v[j];
           esac;
      c[i] = C[i,M];
tel;

-- Dot product
-- Input: v, w: two M vectors
-- Output: s: a scalar
system dotprod : {M | M>1}
          ( v, w : {i | 1<=i<=M} of real)
      returns  (s : real);
var
      S : {i | 0<=i<=M} of real;
let
      S[i] = case
          {| i=0} : 0[];
          {| i>=1} : S[ i-1 ] + v[i]*w[i];
           esac;
      s[] = S[M];
tel;
```