Combined Instruction and Loop Parallelism in Array Synthesis for FPGAs*

Steven Derrien
IRISA
Rennes, France
sderrien@irisa.fr

Sanjay Rajopadhye
IRISA
Rennes, France
rajopadhye@irisa.fr

Susmita Sur Kolay
Indian Statistical Institute
Calcutta, India
ssk@isical.ac.in

ABSTRACT

Compiling perfect, uniform dependence loops to FPGA based co-processors normally yields processor (PE) arrays where a PE executes one instance of the loop body per clock cycle. We develop a transformation framework in which the derived PE can be systematically and automatically pipelined through retiming. We use well known transformations—**skewing** and **serialization**, by which an arbitrary number of registers may be placed at the PE outputs. They are then moved into the PE data-path using standard commerecial circuit retimers. Our experiments (based on performance estimates after place-and-route) have been very encouraging. For a number of examples we have seen dramatic performance improvements: speed increases of an order of magnitude with relatively little (always less than 100%) area overhead.

Keywords

Instruction Level Parallelism, Programmable Logic, Retiming, Regular Processors Arrays

1. INTRODUCTION

High-performance embedded systems are an exploding market, with many applications. These systems often have to deal with very stringent performance (both speed as well as area and power) constraints. Hence, their design is a real challenge, especially in the context of current time to market pressure. Automating the design requires that (i) the application is specified at a high level through standard languages (such as C or MATLAB), (ii) the generated system is formally correct with respect to the initial specification, and (iii) the design tool is flexible enough to allow a relatively wide user-driven design-space exploration.

*Supported in part by IFCPAR project 1802-1: COR-CoP Compilation and Optimization for Reconfigurable Co-Processors

ISSS'01, October 1-3, 2001, Montréal, Québec, Canada.

Critical portions of embedded applications often consist of regular computations generally expressed as nested loops. Hence, the ability to compile loops to specialized parallel circuits should help attain the above objectives and meet performance and power consumption constraints. Several such tools exist in the research community some of which have reached a relative maturity [1, 5]. These tools act as specialized silicon compilers: they generate a structural description (RTL level) of a regular (systolic) array and its associated control from a loop description expressed in a high level specification language. The RTL description can then be used by standard CAD tools to target either an ASIC or an FPGA.

Choosing an adequate target technology is not straightforward. Although some applications still require an ASIC implementation, the advent of dense and cheap FPGAs has made them a viable alternative. However, FPGAs are not as efficient as ASICs, leading to disappointing performance. This is especially true when operations involved in the loop are complex and/or numerous: since traditional array synthesis algorithms implement the loop body as a combinational data-path, the critical path of the resulting design tends to be large, leading to low clock frequency.

However, it is also well known that hand tuned FPGA designs can compete and sometimes even beat ASIC implementations (see [7]). These high performance designs usually take advantage of very fine grain parallelism for which recent FPGA architectures are very well suited: large number of flip-flops, and programmable delay-lines with small area cost. By exploiting similar techniques, we may expect to improve FPGA processing power by pipelining the internal PE structure, so that performance would benefit from intra as well as inter PE parallelism. Our objective is thus to provide tools which will automatically transform a given processor array architecture to take advantage of fine grain pipeline, while preserving its functionality.

In this paper, we propose high-level architectural transformations, which enable very fine grain pipelining in processor arrays. We also propose a heuristic to choose the transformation parameters. Since our transformations are based on the formal foundations of space-time mappings, we can ensure their correctness and incorporate them at a higher level of the design flow, within an array synthesis tool. Our approach is validated experimentally on several examples and shows very encouraging results.

2. BACKGROUND

We now recap the techniques of array processor synthesis,

develop some constraints that we impose on the derived architecture and the corresponding transformations. In doing so we also introduce our notations and conventions.

2.1 Loop parallelization

We only consider a restricted class of loop nests, for which well established methodologies exist. Specifically, we assume perfectly nested loops with uniform flow dependences and statically known bounds. Let I denote the loop domain defined by a set of linear inequalities. Let $\overrightarrow{x}$ denote the loop index vector $[\ x_1 \quad \dots \quad x_n\ ]$. The computation of the loop body is characterized by m data dependence vectors represented as a matrix $D = [\ \overrightarrow{d_1} \quad \dots \quad \overrightarrow{d_m}\ ]$. Consider the loop nest description for $N \times N$ matrix multiplication:

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
  {
    c[i][j]=0;
    for(k=0;k<N;k++)
      c[i][j]=a[i][k]*b[k][j]+c[i][j];
  }
```

The dependencies are the unit vectors $\overrightarrow{i}$, $\overrightarrow{j}$, and $\overrightarrow{k}$, and the iteration space is a three-dimensional cube in the ($\overrightarrow{i}$, $\overrightarrow{j}$, $\overrightarrow{k}$) index space. Following well established methods [8], the parallelization of such a loop nest consists of two steps, namely scheduling and allocation.

The *scheduling* function maps iteration $\overrightarrow{x}$ to time instant $t = \tau(\overrightarrow{x})$, this function must ensure that all data dependencies are satisfied. The schedule is normally an affine function, and we have $t = \overrightarrow{\tau}\,\overrightarrow{x} + c_0$. The condition for the schedule to be valid (with respect to the data dependencies) can be written as $\overrightarrow{\tau}\overrightarrow{d_k} > 0$ where d_k is the k^{th} data dependence vector.

The *allocation* function assigns iteration $\overrightarrow{x}$ to a PE $\overrightarrow{p}$ in the processor index space. As with scheduling, we consider linear allocation functions (specified by a projection vector $\overrightarrow{v}$). They map the n-dimensional iteration space to an $(n-1)$-dimensional processor space. Let π denote this projection matrix, so that $\overrightarrow{p} = \pi\,\overrightarrow{x}$. The whole transformation is expressed as $\overrightarrow{x'} = \Pi\overrightarrow{x} + \overrightarrow{c}$, where $\Pi = \left[\frac{\tau}{\pi}\right]$. For the transformation to be valid, there should be no conflicts (i.e., only one iteration allocated to a PE at any time), and it is well known that this implies that Π should be non-singular.

Note that for each dependence, d_k, the vector $\pi\overrightarrow{d_k}$ gives the corresponding interconnection link in the architecture and τd_k gives its associated delay (in terms of clock cycles).

In addition to the standard constraints described above, we impose two restrictions over π and $\overrightarrow{\tau}$ (for a formal justification please refer to the extented version of this paper [10].

- We only allow unidirectional communications in our array, which translates as $\pi\overrightarrow{d_k} \succeq 0$.
- We only allow nearest neighbor interconnections, which trasnlates as $\forall k,\ \overrightarrow{\tau}\overrightarrow{d_k} \geq \left|\pi\overrightarrow{d_k}\right|$.

2.2 Resulting hardware characteristics

The scheduling and allocation functions yield a processor array where each PE consists of a data-path connected to a set of registers (see Fig. 1 which we shall use as a running example). There are two types of registers in the PE.

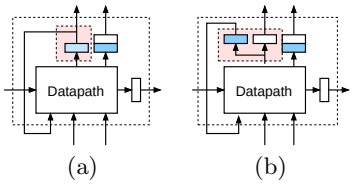


Figure 1: (a) Spatio-Temporal registers in the original architecture (b) modified architecture with non ambiguous register type.

- *Temporal registers* act as local memory within the PE data-path.
- *Spatial registers* connect two PEs along one processor space dimension (for $i = 1 \dots n-1$).

When a register drives both a local (i.e, *temporal*) register and a neighboring PE, It can be considered as both spatial and temporal. In such a case we choose to duplicate this register (by means of retiming) such that the resulting registers are either *spatial* or *temporal* but not both (see figure 1).

The data-path is modeled as a directed acyclic graph of combinational operators, and performs the computations associated with the loop body. It contains n different types of paths (n being the loop dimension).

- A spatial path in the i^{th} dimension ($i = 1 \dots n-1$) is a path starting from any register output or PE input, and driving a spatial register in this dimension.
- A temporal path (i.e., a path in the n-th or the time dimension) is a path starting from any register output or PE input and driving a temporal register.

Note that it is the register at the *end* of the path that determines its dimension.

2.3 Improving Array performance

Obviously, a large combinational path in the data-path is a performance bottleneck, especially for floating point implementations, where FPGAs clock speed implementations can not go beyond 12MHz, although conventional microprocessors attain GHz frequencies. We therefore seek to to pipeline our data-path whenever possible. There are three reasons for this: (i) we expect significant gains in clock speed, (ii) FPGA register resources are generally underutilized, and (iii) in our context, we expect to pipeline aggressively, and tuned to the application at hand.

The pipelining is achieved by "retiming", a technique originally introduced by Leiserson and Saxe [2]. It is a synchronous circuit transformation preserving the behavior of the design while permitting to reorganize the registers of the circuit.

Our goal is here to increase the array achievable clock frequency through retiming. We assume that the PE data-path is represented as a net-list of standard FPGA combinational primitives (each with a predefined delay), so that we can estimate the delay along all combinational paths. Since interconnect delay cannot be estimated prior to the place and route phase, we base our delay estimation on the number of combinational primitives in a path. As in Section 2.2, we partition the PE paths into n subsets, one per dimension.

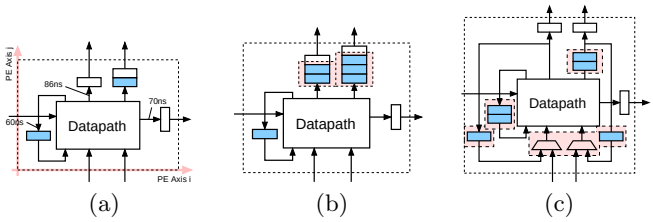


Figure 2: Impact of skewing and serialization. The original PE (a), skewed along the j axis with $\lambda_j = 2$ (b), and then serialized along the j axis with $\sigma_j = 2$.

If we can arbitrarily increase the number of its registers in any dimension, we can pipeline this large combinational path to an arbitrary degree. Of course, the more the registers that we create in a given dimension, the smaller will be the final critical path.

3. TRANSFORMATIONS

We now present two space-time transformations (*skewing* and *serialization*) which allow us to tune the number of registers in the various paths of the PE architecture by affecting both the scheduling and mapping functions.

3.1 Skewing

The **skewing transformation** increases latency between iterations mapped to neighboring PEs. Since this additional latency materializes as registers, which can be used for retiming, we can expect this transformation to allow some clock speed improvement.

The skewing transformation is specified by a vector $\overrightarrow{\lambda}$, its component λ_i being the latency between computations mapped to neighboring PEs along the i^{th} processor index axis;

The modified schedule is then given below. Note that this transformation is only valid for arrays with unidirectional connections (i.e., where $\pi.\overrightarrow{d_k} > 0$, the restriction we imposed in 2.1).

$$\tau'(\overrightarrow{x}) = \tau(\overrightarrow{x}) + \overrightarrow{\lambda}.\pi(\overrightarrow{x})$$

It is easy to show that after skewing, a *spatial* register along a given processor axis i is appended at its input by λ_i temporal registers. This transformation hence increases the number of registers associated with all original spatial paths. However, *skewing* does not affect temporal registers, hence in architectures with temporal loops, *skewing* alone cannot provide registers at appropriate places for retiming. An example of *skewing* is shown figure 2.b.

3.2 Serialization (Clustering)

The **serialization transformation** is well known space-time transformation that has been widely studied [3, 4, 6, 11]. As opposed to classic systolic synthesis, it is non-linear. The key idea is to group PEs derived from the systolisation process into "clusters" in which the original PE's iterations are performed sequentially. The virtual array obtained from the systolisation is hence transformed into a smaller physical array.

Serialization has been used for parallelizing loops on fixed size arrays [6] (also called *active clustering*) or for improving the efficiency of arrays obtained by non-unimodular space-time mappings [3] (called *passive clustering*). Finally, it is also useful for IO bandwidth adaptation [12, 9] and data reuse.

Clustering consists of two coupled sub-problems (i) defining the cluster geometry (ii) specifying the schedule within and between clusters. In the scope of this paper, to ease the derivation and to simplify the control of the serialized architecture, we choose rectangular cluster shape with edges parallel to the communication channels.

Let H be a diagonal rational matrix $\left[\sigma_1^{-1} \ldots \sigma_p^{-1}\right]$, in which σ_i are positive integers. The clustered processor mapping is then $\overrightarrow{p} = \lfloor H\pi\overrightarrow{x}\rfloor$, and the number of iterations to be performed in a cluster is given by $\Pi_{i=0}^{n}\sigma_i$.

Once the cluster geometry is set, iterations within this cluster have to be scheduled. Finding the optimal schedule within a cluster has been studied by several authors, either in the scope of resource constraints (minimizing memory requirements, adequate use of functional units [11]) or to provide simple control logic while preserving the scheduling efficiency [4]. To ease the derivation of the clustered architecture (and hence be able to predict the number and position of all registers), we restrict ourselves to simple scheduling, knowing that (i) cluster boundaries are parallel to the processor space index axes, and (ii) all virtual PEs (i.e., before clustering) are active in a given clock cycle.

For active clustering, these concerns are not relevant. We therefore use a very simple local schedule: iterations are executed in an axis-major order as if our p-dimensional clustering consisted of a sequence of uni-dimensional *serializations* along each dimension of the cluster. Of course, since such serializations are quasi-linear transformations, the order in which they are performed is crucial.

To determine the resulting architecture, we just need to understand the impact of a *serialization* along a given cluster axis. The following rules which can be used to build a serialized architecture

- All temporal registers are duplicated by a factor σ_i. The number of registers associated with the temporal paths (i.e., the depth of the delay line driven by the path output) is thus increased.
- If *serialization* is done along i^{th} processor axis, feed back loops (and their associated multiplexers) are created (as shown in figure 2.b) for all spatial registers along the corresponding dimension.
- The control logic commanding the feed-back loop multiplexer associated to dimension i hence simply consists of a periodically active signal (the signal period is a simple function of the order of serializations and the cluster dimensions).

3.3 Combining all transformations

It is possible to apply these transformations in any order and along any space dimension. However the order in which they are applied has a strong impact on the resulting architecture. This is easy to understand: *skewing* adds temporal registers in all spatial paths, and *serialization* duplicates these temporal registers. If *skewing* is performed before *serialization* its additional registers will be duplicated by serialization, if *skewing* is performed after serialization, its additional register will not get affected. This leads to the

following transformation rules for a $p = n - 1$ dimensional processor space.

- A *skewing* transformation performed along i^{th} cluster axis induces λ_i additional temporal registers on all paths in the i^{th} processor space axis. These additional registers (which are temporal) will hence be duplicated by all subsequent *serializations*. Assuming that all *serializations* are performed in the lexicographic order of the PE space (starting with dimension $\overrightarrow{k}$), we can write the succession of serializations as $\sigma_k, \sigma_{k+1}, \ldots, \sigma_p$. Since all the temporal registers due to *skewing* will be duplicated by each subsequent *serialization*, the number of registers generated in the final architecture is $\lambda_i \Pi_{n=k}^{p} \sigma_i$ (see figure 3.a for the impact of a *skewing* with $\lambda_j = 1$, and figures 3.b and 3.c for the effect of subsequent serializations).

- A *serialization* transformation along the i^{th} cluster axis has two effects: (i) it duplicates all temporal registers by the *serialization* factor σ_i, and (ii) it creates feed-back loops (with one temporal register and a multiplexer) for all spatial paths along the i^{th} processor axis. These temporal registers will not be affected by subsequent skewing, but they will be duplicated by all subsequent serializations. Again, if $\sigma_k, \sigma_{k+1}, \ldots, \sigma_p$ is the sequence of transformations following our initial *serialization* (assuming $i \notin [k, p]$), the number of registers present in the final PE i^{th} axis feed-back loop is $\Pi_{i=k}^{p} \sigma_i$ (eg. the feed-back registers that appeared in 3.b are duplicated by $\sigma_i = 2$ as shown in 3.c)

- When, for a given index i, *skewing* is performed after all serializations, all the λ_i new temporal registers appear outside the feed-back loop induced by the preceding serializations. They can then moved within the data-path if and only if there are enough registers associated to the *serialization* feed-back loop at that time. In other words, since $\Pi_{i=k}^{p} \sigma_i$ denotes the delay-line depth associated with these registers, the number of registers that can be used for pipelining is then $\min(\Pi_{i=k}^{p} \sigma_i, \lambda_i)$ (see figure 3.e for the *skewing* and 3.f for the register factorization).

We now describe some of our assumptions about the order of application of our transformations. First, we will make a distinction between two kinds of skewing. Let λ_i^{pre} denote a dimension-i *skewing* operation performed just before the σ_i transformation and λ_i^{post} denote a *skewing* operation performed after all serializations. We impose the following restrictions.

1. We choose a sequence of axes in the processor space, and for each dimension i in this sequence we execute a pre-σ *skewing* λ_i^{pre}, followed by the σ_i transformation.

2. After step 1 is executed for all dimensions, we perform a succession of post-σ *skewings* by λ_i^{post} for each dimension i.

We now express the number of registers available for pipelining in each dimension: after all the serialization steps, all our initial temporal paths now have $\Pi_{i=1}^{p} \sigma_i$ available registers. We can also formulate the number of registers on the i^{th} spatial path: (i) the combination of pre-σ skew λ_i^{pre} and all its subsequent serializations provides $\lambda_i^{pre} \Pi_{k=i}^{p} \sigma_k$ additional registers along i^{th} spatial axis (ii) the post-σ skew λ_i^{post} provides λ_i^{post} registers, from which only $\min(\Pi_{k=i+1}^{p} \sigma_k, \lambda_i^{post})$ can be used for pipelining. The number of registers along spatial axis i is then $\lambda_i^{pre} \Pi_{k=i}^{p} \sigma_k + \min(\Pi_{k=i+1}^{p} \sigma_k, \lambda_i^{post})$.

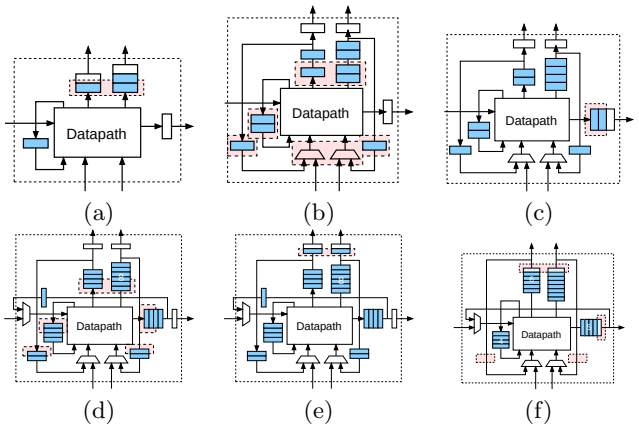


Figure 3: Successive transformations for the PE of Fig. 2.a: (a) Skewing $\lambda_j^{pre} = 1$; (b) Serialization $\sigma_j = 2$; (c) Skewing $\lambda_i^{pre} = 2$; (d) Serialization $\sigma_i = 2$; (e) Skewing $\lambda_j^{post} = 1$; (f) final architecture.

3.4 Choice of tranformations

There is considerable freedom in choosing the order and parameters of the transformations, and we seek one which satisfies the constraints for a given desired throughput. Although several transformations may satisfy these constraints, they are not necessarily equivalent in terms of total register cost. In this paper we use simple pragmatic heuristic to determine a satisfactory, but not necessarily optimal solution. Let d_i denote the given constraints on the minimum number of pipeline stage along dimension i (with d_n for the temporal dimesion).

Generally the cluster dimensions are either set to specific values or (more generally) have lower bounds, so that they can match external (eg. bandwidth and resource) constraints. Hence we will assume that the cluster dimensions are given and are chosen such that $\Pi_{i=1}^{p} \sigma_i > d_n$. The problem is then to choose (i) the order of serializations, and (ii) the values of λ_i^{pre},λ_i^{post}.

For (i), we propose the following: sort space indices in the decreasing order of T_{path_i} and execute the serializations according to this order. This allows the most critical spatial path to benefit most from the additional registers provided by the subsequent serializations.

For (ii), we define $\Delta = \Pi_{k=i+1}^{p} \sigma_k$ if $i < p$, $\Delta = 1$ otherwise. The *skewing* parameters are then given by $\lambda_i^{pre} = \min\left(0, \left\lceil \frac{d_i - \Delta}{\Pi_{k=i}^{p} \sigma_k} \right\rceil\right)$ and $\lambda_i^{post} = \min(d_i, \Delta) - 1$.

3.5 Example

To illustrate the transformation flow, consider the PE whose architecture is shown in figure 2. Assume that we need to meet the following set of inequality constraints over the number of registers required along each dimension of its datapath. We set $d_3 \geq 4$ (4 stages along the temporal axis), $d_1 \geq 5$ (5 stages along processor axis i) and $d_2 \geq 6$ (6 stages

along processor axis j). Given these constraints, the next step is to choose the cluster size. In our case we will only consider constraints over the clock period, hence any cluster shape should be valid provided that the constraint on the number of registers in the temporal paths is satisfied (in other words, $d_3 \geq 4$ stages with $\sigma_i \sigma_j = d_3$). To keep our choice simple we will take $\sigma_i = 2$ and $\sigma_j = 2$. Since the number of required stages along j axis dominates that for i axis, serialization along j is performed before i. Let us determine our transformation parameters:

- For axis $\overrightarrow{j}$, we have $\Pi_{k=i+1}^{p} \sigma_k = 2$. Since we need at least 6 stages along this dimension, we need a pre-serialization *skewing* transformation. Our heuristic gives us the following values: $\lambda_j^{pre} = \lceil \frac{6-2}{4} \rceil = 2$ and $\lambda_j^{post} = 0$.

- For axis $\overrightarrow{i}$, there are no subsequent serializations. Since we need at least 5 pipeline stages along this dimension, we need both a pre and post-skewing transformation along axis $\overrightarrow{i}$ to provide additional registers. We then have $\lambda_i^{pre} = \lceil \frac{5-1}{4} \rceil = 1$ and $\lambda_i^{post} = 1$.

Our resulting architecture is shown in figure 3.f, we see that in all dimensions, the number of registers available for pipeline matches our requirements. The data-path can hence now be pipelined using a retiming tool to move these registers so that the global critical path is reduced down to the estimated target clock period.

4. EXPERIMENTAL RESULTS

Given our framework to exploit fine grain parallelism in processor array compilation, we must study experimentally how these transformations effectively impact (i) clock speed, and (ii) area. To do so, we took several loop nest examples and applied our transformations on them, and then used existing retiming capabilities of commercial CAD tools (Synopsys FPGA compiler II). All PE architectures were described in standard synthesizable VHDL, either using available existing IP modules or the CAD tool arithmetic inference capabilities (as for integer operations). The following loop examples were used as benchmarks.

- Adaptive filter (`DLMS`) for 8 and 16 bit integer and 32 bit floating point.

- Biological sequence comparison (best alignment score) for amino acid (`AA`) and `DNA` sequences

- Matrix multiplication (`MatMul`) for 8 and 16 bit integer and 32 bit floating point.

A key hypothesis in our work is that the area overhead caused by the additional pipeline registers will not impact the total area too strongly. We propose to experimentally validate this hypothesis, by observing the relation between (i) pipeline level, (ii) maximum operating frequency, (iii) PE area cost and (iv) overall raw performance improvement. Our target architecture is a VirtexE-8 from Xilinx. Results, in terms of area use (Virtex Slices) and achievable clock period (in MHz) are those predicted by Xilinx place and route tools and are shown in figure 4.

DLMS-float

Stages	Slices	ns	MHz	ρ
1	1444	85,00	11,76	8,15
2	1413	48,50	20,62	14,59
3	1520	34,00	29,41	19,35
4	1556	27,50	36,36	23,37
5	1587	24,50	40,82	25,72
7	1588	20,00	50,00	31,49
11	1654	18,00	55,56	33,59

DLMS-16bits

Stages	Slices	ns	MHz	ρ
1	353	21,00	47,62	134,90
2	419	13,00	76,92	183,59
3	414	11,50	86,96	210,04
4	421	10,50	95,24	226,22
5	441	9,70	103,09	233,77
7	449	10,00	100,00	222,72
9	452	10,50	95,24	210,70

DLMS-8bits

Stages	Slices	ns	MHz	ρ
1	123	17,40	57,47	467,25
2	147	9,00	111,11	755,86
3	147	8,50	117,65	800,32
4	147	8,50	117,65	800,32
5	147	8,20	121,95	829,60

(a)

MatMul-float

Stages	Slices	ns	MHz	ρ
1	704	85,00	11,76	16,71
2	707	55,00	18,18	25,72
3	713	36,00	27,78	38,96
5	725	24,00	41,67	57,47
7	728	21,00	47,62	65,41
9	748	17,50	57,14	76,39
11	749	16,00	62,50	83,44
13	738	15,50	64,52	87,42
17	766	12,50	80,00	104,44
21	781	13,00	76,92	98,49
25	781	12,50	80,00	102,43

MatMul-16bits

Stages	Slices	ns	MHz	ρ
1	154	17,00	58,82	381,97
2	164	12,50	80,00	487,80
3	194	10,00	100,00	515,46
4	194	9,50	105,26	542,59
5	195	9,00	111,11	569,80
9	206	8,50	117,65	571,10

MatMul-8bits

Stages	Slices	ns	MHz	ρ
1	55	14,00	71,43	1298,70
2	55	7,50	133,33	2424,24
3	70	6,50	153,85	2197,80
4	63	7,00	142,86	2267,57
5	74	6,00	166,67	2252,25

(b)

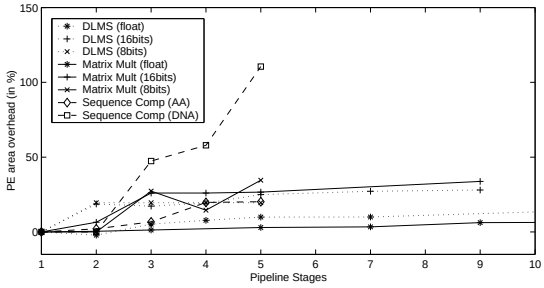
ADN

Stages	Slices	ns	MHz	ρ
1	57	15,00	66,67	1169,59
2	57	13,71	72,94	1279,64
3	84	8,50	117,65	1400,56
4	90	9,50	105,26	1169,59
5	120	7,00	142,86	1190,48

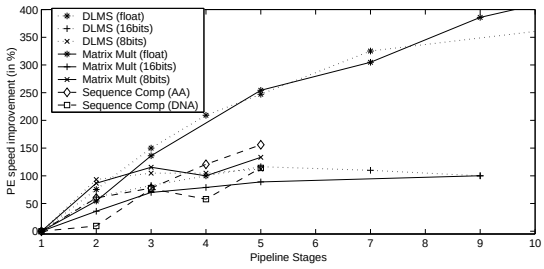
AA

Stages	Slices	ns	MHz	ρ
1	193	32,00	31,25	161,92
2	197	20,00	50,00	254,13
3	206	18,00	55,56	269,69
4	231	14,50	68,97	298,23
5	232	12,50	80,00	344,46

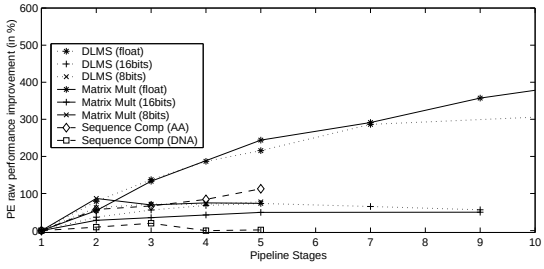
(c)



(d)



(e)



(f)

Figure 4: Experimental results (area, speed and raw performance) for all targeted loop nests

4.1 Area/Clock speed trade-off

Our goal here is to quantify the effective performance improvement induced by our transformation. Specifically, we have shown that deep pipelining requires a lot of registers, which will use more of the available real estate. This leads to a trade-off between area and clock speed: the more registers we use, the better is the clock speed, but the fewer are the PEs that we can fit on the reconfigurable resource. The area overhead due to the pipeline registers is difficult to evaluate analytically, and quantifying the effective performance improvement is not trivial.

In the scope of systolic array implementations, we consider that the array performance is mostly dictated by two parameters: (i) n_{PEs} the number of PEs in the array (i.e., the degree of parallelism) (ii) f_c, the PE speed. We introduce a raw performance metric $\rho = f_c n_{PEs}$ for our architecture. Since n_{PE} is roughly inversely proportional to the PE area (we have $n_{PE} \approx \left\lfloor \frac{A_{fpga}}{A_{pe}} \right\rfloor$ where A_{PE} is the PE area cost and A_{fpga} the target FPGA density), we can also estimate the raw performance by using $\rho^{'} \approx f_c A_{PE}^{-1}$, where A_{PE} is the area cost of the transformed PE. Our results are shown in figure 4. It appears that the raw performance is improved (up to an order of magnitude for complex data-path) but there is also noticeable gain for simpler ones (between 50 to 100%).

We also observed that even by providing a large number of registers, the maximum achievable clock frequency is very dependent on the complexity and regularity of the data-path. We could not get more than 50MHz for a floating point DLMS, while we almost reached 120MHz for a 16 bit integer Matrix multiplication cell. These figures are of course to be compared to the bound for the internal frequency of the target FPGA (130 MHz, as per the device data-sheet).

4.2 Performance limitations

From our experiments, it seems that two factors limit the benefit of using very fine grain parallelism (in other words pipelining down to the logic cell level).

First, it seems that the commercial retiming tool we have been using during our experimentations does not try to reduce the critical path below 5 logic levels. This poses bounds over the minimum achievable combinational path within a pipeline stage, and prevents us from fully exploring the trade-off between area, routability and clock speed for very fine grain pipelined designs. Note that this is a limitation inherent to the CAD tool itself, which could be bypassed, either by using other tools, or by implementing a custom retimer.

Second, routing delays severely impact the minimum achievable clock period. Depending on the regularity (and complexity of the data-path) they represent between 30% (for a 8-bits integer matrix multiplication) and 80% (DLMS with floating point data) of the critical path delay. Since retiming only operates on the combinational delay elements of the path, it cannot take into consideration the impact routing delays. This poses the question of the efficiency of existing retiming algorithm with respect to modern FPGA architectures, in which routing delays are likely to become the performance bottleneck.

5. CONCLUSION AND FUTURE WORK

In this paper, we provided a framework to allow automatic and efficient integration of pipelined data-path for processor array synthesis tools. This framework is based on well-established space-time transformations which ensure the correctness of the pipelined circuit. Although this transformation is not restricted to FPGA implementations, we observed that these types of architectures are very well suited to such transformations which can increase the raw-power of the synthesized architecture by factors up to 6.

Although the results are very encouraging we believe that there is need to further explore the trade-off involved with very fine grain pipelined designs. Specifically since routing delay is likely to have a major influence on such designs, we believe that there is some need for a layout-aware retiming tool which would be able to compensate the routing influence by exploiting placement information. This is the subject of our ongoing investigations.

6. REFERENCES

[1] The MMAlpha Environment. COSI Website, IRISA http://www.irisa.fr/COSI/ALPHA.

[2] J.B. Saxe and C.E. Leiserson. Optimizing synchronous systems. In *J. VLSI and Computer systems*, 1983.

[3] A. Darte. Regular Partitioning for Synthesizing fixed-size Systolic Arrays. In *INTEGRATION, the VLSI journal*, 1991.

[4] A. Darte et al. A Contructive Solution to the Juggling Problem. In *International Conference on Application Specific Processor Arrays (ASAP)*, 2000.

[5] R. Schreiber et al. High-level Synthesis of Non-programmable Hardware Accelerators. In *IEEE conference on Application Specific Array Processor*, 2000.

[6] E.F. Depreterre P. Dewilde and J. Bu. A Design Methodology for Partitioning Systolic Arrays. In *IEEE conference on Application Specific Array Processor*, 1990.

[7] C. Patterson. High performance DES encryption in Virtex FPGAs using JBits. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Press, 2000.

[8] P. Quinton. Automatic Synthesis of Systolic arrays from Recurrent Uniform Equations. In *International Conference on Computer Architecture*, pages 208–214, 1984.

[9] S. Derrien S. Rajopadhye and S. Sur-Kolay. Optimal partitionning for FPGA based regular array implementations. In *IEEE PARELEC'00*, August 2000.

[10] S. Derrien S. Rajopadhye and S. Sur-Kolay. Combining Instruction and Loop Level Parrallelism for FPGAs. *IRISA Research report N1376*, 2001.

[11] L. Thiele J. Teich and L. Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Contrained Resources. In *International Conference on Application Specific Processor Arrays (ASAP)*, 1996.

[12] Renate Merker and Uwe Eckhardt. Co-Partitionning - A Method for Hardware/Software design for scalable Systolic Arrays. In *Reconfigurable Architectures, ITPress*, 1997.