Increasing pipelined IP core utilization in Process Networks using Exploration

Claudiu Zissulescu, Bart Kienhuis, Ed Deprettere

Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
PUBLISHED IN THE PROCEEDINGS OF FPL'04, ANTWERP 2004

Abstract. At Leiden Embedded Research Center, we are building a tool chain called *Compaan/Laura* that allows us to do fast mapping of applications written in Matlab onto reconfigurable platforms, such as FPGAs, using IP cores to implement the data-path of the applications. A particular characteristic of the derived networks is the existence of selfloops. These selfloops have a large impact on the utilization of IP cores in the final hardware implementation of a PN, especially if the IP cores are deeply pipelined. In this paper, we present an exploration methodology that uses feedback provided by the Laura tool to increase the utilization of IP cores embedded in our PN network. Using this exploration, we go from 60MFlops to 1,7GFlops for the QR algorithm using the same number of resources except for memory.

1 Introduction

To better exploit the reconfigurable hardware devices that are coming to market, a number of companies like AccelChip and Celoxica and research groups around the world [5, 1] are developing new design methodologies to make the use of these devices more ubiquitous by easing the way these devices are to be programmed. At Leiden Embedded Research Center, we are developing a design methodology that allows us to do fast mapping of applications (DSP, imaging, or multi-media) written in Matlab onto reconfigurable devices. This design methodology is implemented into a tool chain that we call *Compaan/Laura* [9]. The Compaan tool analyzes the Matlab application and derives automatically a parallel representation, expressed as a Process Network (PN). A PN consists of concurrent processes that are interconnected via asynchronous FIFOs. The control of the input Matlab program is distributed over the process and the memory is distributed over the FIFOs. Next, the Laura tool synthesizes a network of hardware processors from the PN. Laura derives automatically the VHDL communication structure of the processor network as well as the control logic (interfaces) of the processors needed to attach them to the communication structure. The computation that has to be performed in every processor is not synthesized by the Laura tool. Instead, the tool integrates commercial IP cores in every processor to realize the complete computation.

When pipelined IP cores are used in a PN processed by Laura, new data can be read without waiting for the finalization of the current computation. Therefore, a network is seen as a big pipelined system. To maximize the throughput of a network, we need to

carefully consider how to couple the pipelines of the processors in a network. In this coupling, one particular characteristic plays an important role. This is the existence of selfloops. A *selfloop* is a communication channel that sends data produced by a processor to itself. Selfloops appear very frequently in PNs and they impact the utilization of IP cores thus have a great impact on the overall performance of PNs. A pipelined IP core works at maximum throughput if the network provides as many independent operations as the depth of the pipeline.

In this paper, we present how we can perform a design space exploration to increase the utilization of IP cores. Using exploration, we change in a systematic way the size of the selfloops as they are a measure of independent operations. This measure is obtained at compile-time by extending Laura with a *Profiler* option. This provides systematic hints for further exploration. Using this profiler, we performed an exploration that is presented at the end of this paper. In this exploration, we realizes a throughput improvement from 60MFlops to 1,7GFlops for a QR algorithm, using the same number of processors.

2 The Compaan/Laura tool chain

In general, specifying an application as a PN is a difficult task. Therefore, we use our *Compaan* Compiler [6] that fully automates the transformation of a Matlab code into PNs. Subsequently, the *Laura* [11] tool takes as its input this PN specification and generates synthesizable VHDL code that targets a specific FPGA platform. The Compaan and Laura tools together, realize a fully automated design flow that maps sequential algorithms written in subset of Matlab onto reconfigurable platforms. This design flow is shown in Figure 1.

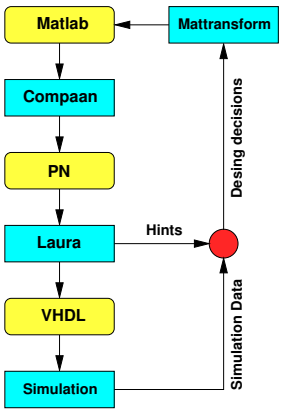

Fig. 1. The Compaan/Laura tool chain

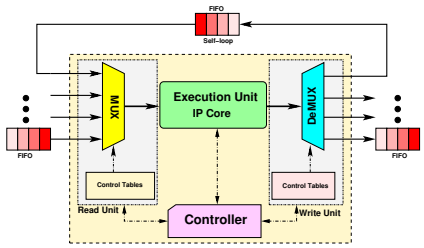

Fig. 2. The Virtual Processor model

In the first part of the design flow, an application specification is given in Matlab to Compaan to be compiled to a PN representation. The applications Compaan can handle

are parameterized static affine nested loop programs, which can be described using a subset of the Matlab language. In the second part of the design flow, Laura transforms a PN specification together with predefined IP cores into synthesizable VHDL code. The IP cores are needed as they implement the functionality of the functions calls used in the original Matlab program. In the third part of the design flow, the generated VHDL code is processed by commercial tools to obtain quantitative results. These results can be interpreted by designers, leading to new design decisions. This is done with the help of Mattransform tool [8] that performs a source-to-source transformation on the Matlab code. By rewriting the Matlab code, we can explore different mappings. When an obtained algorithm instance meets the requirements of the designer, the corresponding VHDL output is synthesized by a commercial tool and mapped onto an FPGA platform. Recently, we have extended Laura with a 'Profiler' option to provide hints at compile time that can be used by Mattransform to rewrite the application to increase the performance of the input algorithm.

To show the relation between a function call in Matlab program and an IP core, we need to explain the way Laura realizes the functionality of the function call. This functionality is wrapped and executed in a corresponding processor of the derived PN. To implement the PN in hardware, Laura uses the notion of Virtual Processors and hardware communications channels (i.e. FIFOs).

A Virtual Processor is composed of four units: a *Read* unit, a *Write* unit, an *Execute* unit, and a *Controller unit*, as shown in Figure 2. The Execute unit is the computational part of a virtual processor. The Read unit is responsible for assigning all the input arguments of the Execute unit with valid data. The Write unit is responsible for distributing the results of the Execute unit to the relevant processors in the network. The Controller of the Virtual Processor synchronizes all the units of the processor. The Read unit and the Write unit can block the next execution when a blocking-read or a blocking-write situation occurs, thereby stalling the complete processor. A blocking-read situation occurs when data is not available at a given input port. A blocking-write situation occurs when data cannot be written to a particular output port. In the Execute unit an IP core is embedded. This IP core implements the functionality specified in the original Matlab code. The controller automatically fires the execution unit when data is available.

3 Problem Definition

The task to interpret the quantitative results obtained from VHDL traces can be very difficult in a complex PN network. Therefore, we want to guide the designer using the tool chain with indications that are obtained from the analysis of the network and prior knowledge on the IP cores.

Laura's Virtual Processor is a pipelined processor. Therefore, the filling and flushing of the pipeline reduces the throughput below the maximum level achievable. One of the problems that affect the efficiency of our designs is given by the data availability. Data availability is affected when selfloops are involved. When a processor has a selfloop, it can happen that the processor want to read data from the selfloop, but that data is still in the pipeline. Because the data is in the pipeline, the processor has to wait, reducing the efficiency of the pipeline of the IP core.

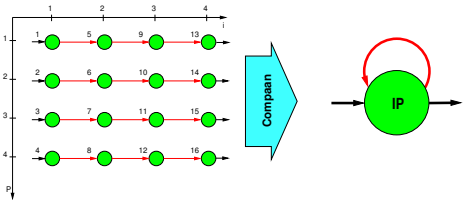

Fig. 3. Simple Example

Let us consider a simple example by looking at the dependency graph (DG) in Figure 3. This DG is folded by Compaan into a single processor that has a single selfloop, which corresponds to data dependencies in the i direction. In Compaan, the nodes in the DG are scheduled. In Figure 3, the schedule is given by the p direction (e.g. 1,2,3,4,..,14,15,16). It is obvious that the operations 1,2,3,4 are independent, i.e. there is no arrow between them and, therefore, no data dependency. Therefore, the processor has to store the result of these operations into a temporary memory until they are consumed by the following 4 independent operations (e.g. 5,6,7,8). In our case, this memory is a selfloop, implemented as a FIFO channel. Hence, the necessary size for the selfloop is given by the number of independent operations, which is 4 in the case given in Figure 3. If we consider an alternative schedule given by the i direction (e.g. 1,5,9,13,..,12,16) the size of the selfloop FIFO is one. In this case, the data is immediately consumed by the processor after being produced.

Now, let us assume that the processor is pipelined and its pipeline depth is equal to 4. For the first schedule example, the pipeline achieves the maximum throughput because all the pipeline stages are filled with independent operations. The processor takes data either from a source (i.e. the first four iterations) or from itself (i.e. the remaining iterations). In the second schedule example, the pipeline is underutilized because the data generated by the processor is not yet available at its input due to the pipelining. This case should be avoided as the pipeline of the IP core is used for only 25% of its capacity.

4 Solution Approach

The number of independent operations that can be mapped onto a processor is a key metric to obtain efficient implementations. Hence, we developed a procedure that reports the number of independent operations that are mapped onto a Virtual Processor that has selfloops. We call this the *Profiler* of Laura. The procedure computes the size of a selfloop, which represents the amount of independent operations for a processor. The number is reported back to the designer who may take the necessary decisions to improve the quality of the analyzed design. Design decisions such as retiming (i.e. skewing), loop swapping, adding more streams of the same problem and unfolding can be applied to control the amount of independent operations that are mapped onto a processor. Using the procedure to compute the size of the selfloop and the given design

decisions, we can close the loop in Figure 1 needed to perform a design space exploration to improve the efficiency of an input algorithm mapped into a reconfigurable platform.

5 Detecting the selfloop size

Normally, we cannot determine the size of a FIFO in a PN as we only specify the partial order between the processors. To determine the size of each FIFO would require a global schedule of the network, which cannot be easily determined. The selfloop is, however, a special case as this kind of communication channel starts and ends on the same processor and the writing and reading to/from this channel respects the internal schedule of the processor. We exploit this special case to determine the size of the FIFO channel.

We use the polytopes model [2] to represent mathematically a statical nested for loop program that is taken as input by our tool chain. Each processor is characterized by a polytope that represents its iteration space. The original for-loops gives us a schedule on this iteration space. This schedule is implemented in the Read and Write units of the Virtual Processor and its responsibility is to communicate the right data with other processors in the network.

The order in which write and read operations are performed over the selfloop depends on the local schedule of the processor. The selfloop size is given by the number of write operations that are done before the first read operation is performed, which defines the *run in* period. Due to the schedule of a processor, it is guaranteed that after the first read is done, other read operations will follow at constant time intervals, which defines the *steady stage*. In this stage, the read operations are interleaved with the write operations. The steady stage is followed by the *run out* period, which is characterized by performing only read operations. Hence, to compute the selfloop size, it is necessary to determine how many write operations were made in the run in period. This is equivalent to counting the number of integral points in the polytope that defines the run in period. To compute the points, we made use of the Ehrhart theory [4].

A processor can have one or more selfloops and each selfloop can have a different size. The minimum size out of all the processor selfloops represents the minimum amount of independent operations that are mapped on the IP core. Therefore, we consider this minimum size for our efficiency analysis.

6 Increasing the throughput of a Virtual Processor

The Virtual Processor is a pipelined model; reading, executing, and writing are done in a pipeline fashion, as shown in Figure 2. To process a token, the read unit requires τ_{read} cycles to fetch the token. The execute unit requires $\delta_{pipeline}$ cycles in the IP core to execute. The write unit requires τ_{write} cycles to write a token into an output FIFO. To achieve 100% utilization for the IP core in an ideal network, the required number of independent operations Σ that needs to be mapped on the IP core is given by Equation 1.

$$\Sigma = \delta_{pipeline} + \tau_{write} + \tau_{read} \tag{1}$$

6.1 Pipelined IP cores and independent operation

To optimize the IP efficiency, we need to look at the number of pipeline stages and the size of the FIFO selfloop. If the size of the FIFO is smaller than the number of pipeline stages, then we can increase the number of independent operations. This can be done in two ways. The first way is to apply an unimodular transformation (e.g., skewing, loop swapping) on the local schedule of the Virtual Processor local schedule that contains the IP core. The second way is to add

$$P = \Sigma - Size_{FIFO} \tag{2}$$

independent streams of the same problem to be processed in a pipeline fashion by our network (i.e., data-stripping). This gives us the necessary independent operations in the processors for an optimal throughput. These P independent stream must be interleaved in a optimal way. We relay on Compaan to do this for us by rewriting the original Matlab code with one extra for-loop that has as upper bound the value P. This loop must be the inner most loop to achieve the interleaved execution of the streams. If the size of the selfloop is larger than the number of pipeline stages, then the core is overloaded with independent operations. We can take advantage of this surplus of independent operation by unrolling this processor to increase the number of parallel running resources. An unrolling operation balances the workload on two or more identical processors, and may improve the performance for the entire algorithm. The proposed procedures of independent streams and unrolling do not guarantee an efficiency of 100% of the IP core. This is because the core efficiency depends also on the data dependencies between different processors.

6.2 Single cycle IP cores and one independent operation

A special case exists when the IP core is not pipelined at all. If a selfloop of size 1 is mapped on the IP core, we have an instance of the classic case of *data hazard*. In this situation the selfloop is replaced with a simple wire between the read and write unit of the virtual processor. This technique is very beneficial, as the wire increases the throughput of the processor and requires less hardware resources.

7 QR: A case study

To explain the hints the profiler implemented in Laura presented and how that effects design decisions, we now consider the QR case, which is widely used in signal processing applications [10]. The Matlab code for the QR algorithm that can be processed by Compaan is presented in Figure 4. It shows two function calls (*bcell* and *icell*) surrounded by parameterized for-loops. Compaan will generate a PN based on the for-loops and the variables passed on to the function calls. What happens in the *bcell* and *icell* is irrelevant to Compaan, but not for Laura. The network we obtain is given in Figure 4. Observe that each function call becomes a process in Compaan and that both the *bcell* and *icell* processes have selfloops.

The *bcell* and *icell* are realized by integrating a **bcell** and **icell IP** core on the execute unit in a virtual processor (See Figure 2) for the *icell* and *bcell* function call. The **bcell** and **icell** IP cores are pipelined and have 55 and 42 stages, respectively. To arrive to an optimal QR implementation each core must, therefore, have 57 and 44 independent operations, according to Equation 1. We look at QR with the parameters set to the typically values of **N = 7** and **T = 21**. For the case presented in Figure 4, our profiler reports for the smallest selfloop of **icell** a size five and for **bcell** a size 1. Given the deep pipelines of the **icell** and **bcell** IP cores, both of them are underutilized and the profiler recommends either to explore a space of additional independent QR streams from 1 to 57, or to perform a skewing operation.

First, we choose to explore the design decision of adding independent streams, and then we evaluate the design decision of skewing. For each experiment, we derive a VHDL representation of the algorithm using the Compaan/Laura tool chain that we can simulate to obtain quantitative data. The execution duration(No_{cycles}) for each experiment is measured in cycles. Each of the **icell**(No_{icell}) and **bcell**(No_{bcell}) operations contain 11 and 16 floating point operations, respectively. The average speed of our QR network mapped onto an FPGA platform is 100Mhz. We use the next formula to compute how many million floating point operations per second (MFLOPS) can be achieved in each experiment.

$$\Delta = \frac{No_{icell} * 11 + No_{bcell} * 16}{No_{cycles}} * 10^6 \tag{3}$$

7.1 QR: Adding more streams

As we input the original QR algorithm in the chain, the profiler reports that we have to add 57 independent streams to the **bcell** IP core and 40 independent streams to the **icell** IP core by applying Equation 2. Hence, we choose to explore the space made by running 1, 10, 20, 30, 40 and 57 independent streams. In this case, the stream represent complete instances of the QR algorithm. The results of this exploration are given in Figure 5.

We observe that the saturation point is marked by the running 40 QR instances. At this point, the profiler reports 4.54 times more independent operations than the **icell** can handle. Therefore, we choose to unfold the **icell** twice and four times respectively.

```
for k = 1:1:T,
  for j = 1:1:N,
   [r(j,j), rr(j,j), a, b, d(k) ] =
    bcell( r(j,j), rr(j,j), x(k,j), d(k) );
   for i = j+1:1:N,
    [r(j,i), x(k,i)] =
     icell( r(j,i), x(k,i), a, b );
   end
  end
end
```

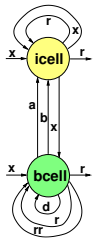

Fig. 4. The QR matlab code and the equivalent PN

This leads us to 1764MFLOPS and 1767MFLOPS, respectively. The first transformation gives us an additional 81MFLOPS, while the second one only a difference of 3MFLOPS. It is obvious that the second operation is not as successful as the first one. We should also indicate that although we didn't use more IP cores, we obtained the higher throughput at the expense of more memory.

7.2 QR: Skewing

A second option to increase the number of independent operations in each processor is to skew the algorithm in Figure 4. After applying this transformation, the profiler indicates 7 independent operations in **bcell** and 21 in **icell**. However, these numbers are not sufficient to fill the entire pipeline.

To achieve a higher throughput for our experiment, we must either increase the dimension of the input problem or add more independent streams, i.e., QR instances. For the first case the required parameters will be **N = 57** and **T = 44**. For the second one, a space of 10 independent QRs is suggested by the profiler. We choose to explore 2, 4, 6, 7, and 10 independent streams. The results are shown in Figure 6. The first bar from the graph represents the throughput of the original QR, the second bar is the skewed version and the following bars are the values obtained for the skewed QR algorithm with increasing number of independent QR streams.

We choose to unroll the QR algorithm running 4 QR instances. In this case, the profiler reports a 181% utilization for the **icell**. Unrolling the **icell** core twice gives us only 6MFLOPS more throughput compared with the non-unfolded algorithm. This shows us that the decision was not so successful. The reason for this is that the unfolded **icell** cores work in a mutual exclusive fashion, nulling the effect of the unrolling operation.

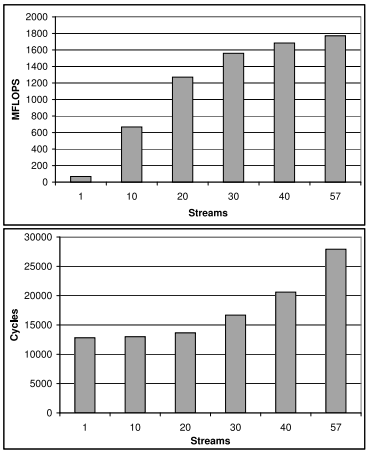

Fig. 5. QR experiments with additional streams

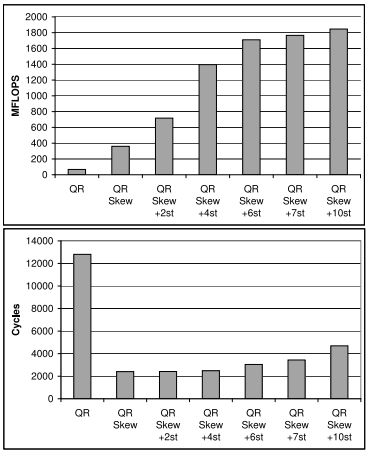

Fig. 6. QR experiments with skewing and additional streams

7.3 Discussion

Based on the values given in Figure 6, skewing the algorithm gives us from the beginning an important amount of independent operations that allows to obtain a higher computational throughput without adding independent streams. The skew transformation fills the pipeline with parallel data that belongs to the same input problem. However, it may be that the skewing transformation doesn't achieve the maximum throughput of the cores and, therefore, solutions such as increasing the dimensions of the input problem (i.e. changing the values of **T** and **N**) or adding more independent streams may be required.

The existence of parallel operations that belongs to the original algorithm helps the designer in achieving the maximum throughput with minimum amount of independent streams added. These operations are exposed to the architecture through the skewing transformation. In the non-skewed version the saturation of our architecture is reached around 40 streams, while in the case of the skewed version the saturation point is given by 6 streams. In real life, it is more likely that only a small number of independent instances of an algorithm need to be computed at the same time. Therefore, the skewed version is more appealing to start with in any design space exploration. The overloading of an IP core with many streams can be solved either by increasing the clock speed for that particular processor (i.e., using various clock domains) or by unrolling it. In our experiment, we showed that applying high-level transformations (e.g., skewing, unfolding, data stripping) as offered by the Laura profiler as hints leads to an increased throughput. Nevertheless, quantitative simulation data is needed to assess its final usefulness.

8 Related Work

There are numerous researchers that recognized the need for fast performance estimation to guide the compilation of a high-level application for deriving alternative designs. The PICO project [5] uses estimations based on the scheduling of the iterations of the nested loop to determine which loop transformation lead to shorter scheduling time. The MILAN project [1] provides a design space exploration and simulation environment for System-on-Chip architectures. MILAN uses simulation techniques to derive estimates used in the evaluation of a given application. In [3] the authors use an analytical performance model to determine the best mapping for their processor array. An analytical model is used also in [7] to derive performance and area for a given nested-loop program. All these projects are focused on synchronous systems, and, therefore, a global schedule of their system can be derived at compile time. However, we derive estimations regarding the throughput of deeply pipelined IP cores in the absence of a global network schedule. In [8], we have shown already that we can apply loop transformations to increase the performance of our networks.

9 Conclusions and future work

A particular characteristic of the derived networks we obtain from running the Compaan/Laura tools, is the existence of selfloops. These loops have a large impact on the

utilization of the IP cores and in the final hardware implementation of a KPN. This is especially the case when the IP cores are deeply pipelined. To improve the efficiency, the designer has to make design decisions like skewing, unrolling, loop swapping and data stripping. To help the designer to in making these decisions, we have implemented the profiler in Laura. The profiler uses manipulation of polytopes to compute at compile time the size of selfloops. This size is indicative for the number of independent operations available in an algorithm. The computed hints provides by the profiler, help to steer design decisions. Doing this in a iterative manner, a designer can explore options to improve the throughput of a process network. In this paper, we have shown for the QR algorithm that we could improve the performance from 60MFlops to 1.7GFlops by using the hints from the profiler. Using the hints, we could improve the utilization of mapping the QR algorithm on a FPGA with deeply pipelined IP cores by a factor of 30, using the same IP cores albeit at the expense of more memory.

To improve the efficiency, the designer has to make design decisions. These operations can be expressed at the Matlab level using the Mattransform tool. Currently, the hints provided by the profiler needs to be manually expressed. Future work is to make a connection between the Mattransform tool and the hints from the profiler.

References

1. A. Bakshi, V. K. Prasanna, and A. Ledeczi. Milan: A model based integrated simulation framework for design of embedded systems. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 82–93. ACM Press, 2001.
2. U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993.
3. S. Derrien and S. Rajoupadyhe. Loop tiling for reconfigurable accelerators. In *Field-Programmable Logic and Applications, 11th International Conference, FPL 2001, Belfast, Northern Ireland, UK*, volume 2147 of *Lecture Notes in Computer Science*. Springer, 2001.
4. E. Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*. Birkhäuser Verlag, Basel, international series of numerical mathematics vol. 35 edition, 1977.
5. V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. Pico: Automatically designing custom computers. *Computer*, 35(9):39–47, 2002.
6. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.
7. K. S. Shayee, J. Park, and P. Diniz. Performance and area modeling of complete fpga designs in the presence of loop transformations. In *Field-Programmable Logic and Applications, 13th International Conference, FPL 2003, Lisbon, Portugal, 2003, Proceedings*, Lecture Notes in Computer Science. Springer, 2003.
8. T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *10th Int. Symposium on Hardware/Software Codesign (CODES'02)", pp. 7-12, Estes Park, Colorado, USA*, May 6-8, 2002.
9. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004.
10. R. Walke and R. Smith. 20 gflops qr processor on a xilinx virtex-e fpga. In *Advanced Signal Processing Algorithms, Architectures, and Implementations X*, volume 4116, 2000.

11. C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 1-3 2003.