Framework for Synthesis of Virtual Pipelines *

Srinivasan Dasasathyan, Rajesh Radhakrishnan and Ranga Vemuri

Department of ECECS, ML0030
University of Cincinnati, Cincinnati, OH 45221.
E-mail: {sdasasat,rradhakr,ranga}@ececs.uc.edu

Abstract

Virtual Pipelining *allows designs of arbitrary size to execute on finite sized FPGA devices. It allows pipelined designs to be efficiently configured on a FPGA by overlapping the reconfiguration time of a pipeline stage with the execution time of previous pipeline stages. This technique produces performance improvement up to an order of 5 versus a non-pipelined execution of a design. We extend this principle for handling large designs that were previously too large to fit on an FPGA. This paper presents a framework for automatically synthesizing virtual pipelines on an Virtex FPGA. We also suggest criteria for extending our approach to non-Virtex FPGAs.*

1. Introduction

Researchers have proposed approaches to solve the important problem of increasing the throughput of a design executing on FPGAs. An important factor in throughput calculation for designs executing on FPGAs is the *reconfiguration* time. Reconfiguration time can be decreased by combining dynamic or run-time reconfiguration with partial reconfiguration where a part of the whole chip is configured while the remainder is still executing. Researchers have proposed approaches to reduce the dynamic reconfiguration time. Noteworthy among them is the Virtual Pipelining approach proposed by Goldstein et al. [4].

1.1 Virtual Pipelining

Virtual pipelining configures a pipelined design by configuring the pipeline stage incrementally as the pipeline gets filled. This is in contrast to other approaches where the entire FPGA is configured with all the stages. Also reconfiguration of a stage is overlapped with the execution of stages that have already been configured. As the configuration of stages happens concurrently with the execution of other stages, there is no loss in performance due to reconfiguration. The above feature is called *incremental reconfiguration* [6]. Another feature of virtual pipelining is *Hardware Virtualization*, which allows design of any size to be mapped efficiently on a single device. This is achieved by swapping out the stage from the device that has executed most number of cycles and configuring that portion of the chip with the next stage that needs to be executed.

As no concrete implementation of virtual pipelining theory exists, we show that the Virtex [7] architecture is suitable for implementing virtual pipelines and provide a design flow to execute the pipelined designs on a Virtex-based boards.

The paper is organized as follows: The motivation for choosing Virtex and its architecture is explained in Section 2. The framework to generate the partial bitstreams for the individual pipeline stages is explained in Section 3. The design flow used to test the framework and is explained in Section 4. Section 5 and 6 present the results and conclusions.

2 Motivation to Target Virtual Pipelines on Virtex-Based FPGAs

To execute virtual pipelines on a FPGA, we can either design a FPGA which has a PipeRench style architecture [4] or look for FPGA that can satisfy the following requirements:

- Support partial reconfiguration and dynamic reconfiguration.
- Time taken to reconfigure a stage should be small.
- Provide enough memory and logic elements to implement pipeline registers and stages.

One such device which satisfies all the requirements is Xilinx's *Virtex*-based FPGA [7]. The Virtex FPGA is divided into a number of columns and each column can be independently reconfigured without affecting the other. Each column has number of Configurable Logic Blocks (CLB)

*This research is supported in part by US Air Force, Wright Labs, WPAFB, under contract number F33615-97-C-1043

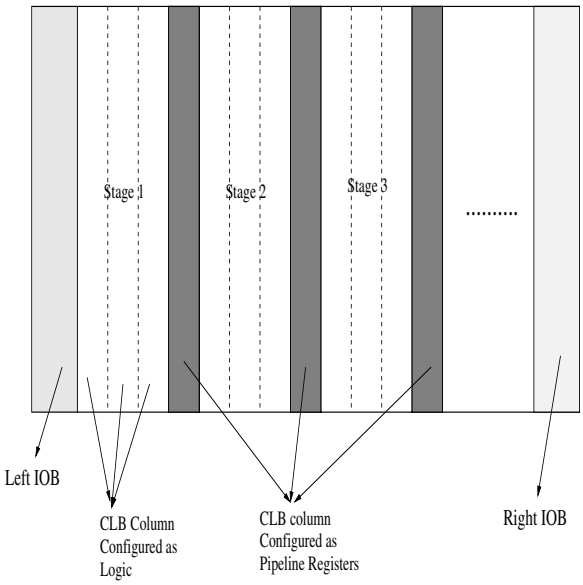

Figure 1. *Pipelined Designs on Virtex*

and has programmable logic and flip-flops. The logic inside the CLB can be configured as stages of a pipeline and the flip-flops as registers. A sample pipelined design using the CLB columns in virtex as pipeline stages and flip-flops as registers is shown in figure 1

To configure individual pipeline stages while executing others, we need to generate a partial bitstream for the next stage that configures.

3 Virtual Pipelining Framework (VPF)

In order to use the virtual pipelining technique to commercial FPGAs, it is necessary to generate partial bitstreams to configure the pipeline stages incrementally. Generating partial bitstreams allows to dynamically reconfigure the FPGA, so that there is a overlap of reconfiguration and execution time. We proposed a flow for generating partial bitstreams by (1) partitioning the design in to combination of pipeline stages (called as partial designs) (2) perform guided placement and routing for these partial designs based on the information from the previous partial designs (3) generate partial bitstreams by taking the difference of bitstreams between two successive partial designs. The framework for generating partial bitstreams is explained in Figure 2.

The flow consists of:

1. *Partial Design Generation (PDG):* Partial design is a design constructed from the input design which reflect the state of the FPGA device during every clock cycle. The partial design changes every clock cycle. It repeats after a interval which depends on number of virtual stages (no of stages in the pipelined design) and physical stages (no of stages that the FPGA can hold at a time) present. A number of partial designs are generated from the input design. The PDG is discussed in Section 4.4.

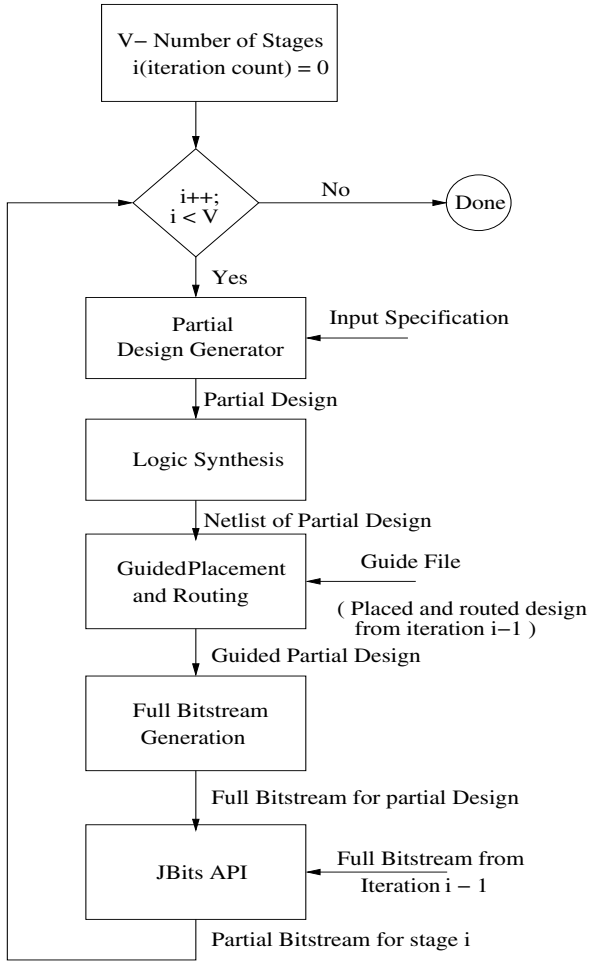

Figure 2. *Framework for Virtual Pipelining*

2. *Guided Placement and Routing:* The current partial design is placed and routed using the information from the previous partial design. Thus all the common stages between two successive partial designs are placed and routed identically. The full bitstream for the current partial design is generated.

3. *Partial bitstream Generation:* The partial bitstream of a pipeline stage can be obtained by taking the difference between two successive partial designs. Using the partial reconfiguration API [8] the difference bitstream to configure a particular pipeline stage alone can be generated.

Steps 1 to 3 are repeated until all the partial bitstreams

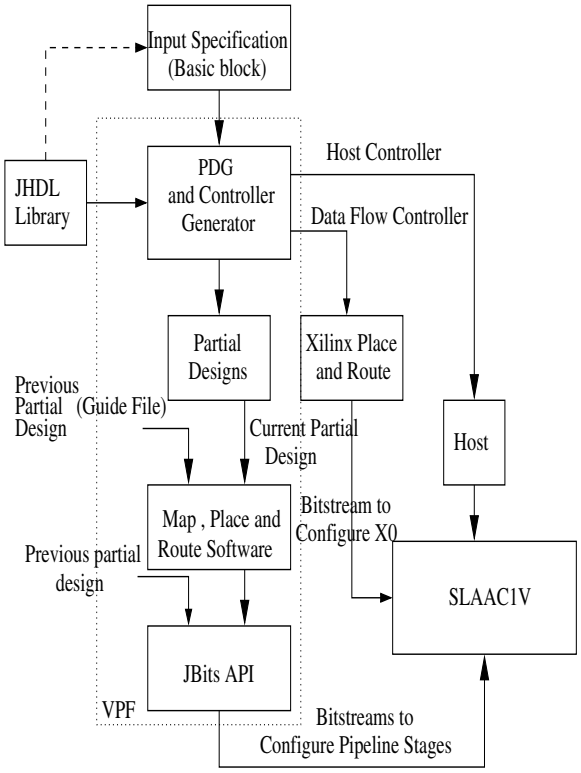

Figure 3. *Using VPF with the SLAAC-1V Board*

for all partial designs are generated.

4 Design Flow for Synthesis and Execution of Virtual Pipelines

We used SLAAC-1V board [2] to test our framework. The overall design flow for synthesizing and executing virtual pipelines on SLAAC-1V is shown in Figure 3. The input design specified as basic blocks uses operations which have corresponding implementations in JHDL [1], which is target HDL to which the design is translated. The translator/partial design generator(PDG) takes this design and splits it into a number of partial designs in JHDL. After the simulation, each of the partial designs is synthesized to obtain the net-list. The generated net-list is mapped, placed and routed on the Virtex device using Xilinx M1 tools. The translator also produces a pin constraint file which assigns pin locations to the input and output ports of the partial design. After this point the partial bitstream generation flow is used to obtain the partial bitstreams of pipeline stages.

The following components constitutes our design flow on SLAAC-1V (1) the basic block input specification (2) the intermediate HDL language JHDL (3) Partial design generator(PDG) (4) Data flow controller generation and (5) Host controller generation.

4.1 Input Specification : Basic Block

We used a file format that specifies a design in terms of basic blocks [5].

4.2 Intermediate HDL : JHDL

The input design specified using basic block is translated into a JHDL for simulation and synthesis. JHDL [3] is a set of FPGA CAD tools that allow the user to design the structure and layout of a circuit, debug the circuit in simulation, net-list and interface with back-end tools for physical synthesis, and so forth. JHDL not only provides structuring and layout of circuits but also a simulation and synthesis environment. The choice of JHDL was natural, because the level of control required to place the logic in a way that is suitable to our application was only provided by JHDL. Also for guided routing to work, the nets in the two successive partial design with common pipeline stages should have same names. Most of the commercial synthesis tools insert the net names randomly. Using JHDL it is possible to give names to nets that user desires.

4.3 SLAAC-1V Board Architecture

SLAAC-1V, the virtex based board shown in Figure 4 was used to test the frame work. The board contains three (X0, X1, X2) Virtex XCV1000-FG680 FPGA, the largest in Virtex family. The board contains rich interconnect for the interaction between all the three FPGAs. The interconnects are fixed and programmable. The fixed interconnect between the FPGAs are 72 bits wide, and form a ring (hence called ring interconnect). The programmable interconnect is a crossbar that is shared between the three FPGAs and is 72 bits wide. There are 10 36x256K ZBT SRAMs, out of which 5 can be accessed by X1 (4 memories at a time), 5 can be accessed by X2, and X0 can access all 10 but only 2 memories at a time. On board switches swap one of X0's memories with that of X1, and the other with X2.

4.4 Partial Design Generator (PDG)

As mentioned previously, partial designs (PD) are identified in the i/p specification. Each PD can be constructed using the component from the JHDL library. The JHDL library was developed keeping in mind the most frequently used components. The library is also characterized with size of each components so that the PDG can use it to place the design efficiently on the chip. As in virtex, the reconfiguration time is proportional to the number of columns that the design occupy, the placement of the components is done column-wise in order to reduce the reconfiguration time.

4.5 Data Controller Generation

The data controller is required to control the flow of data to and from the SLAAC-1V board. It reads the data from the memory and provides inputs to the pipeline stages. The controller reads the processed data from the pipeline stages and writes it to memory. The controller has a state machine

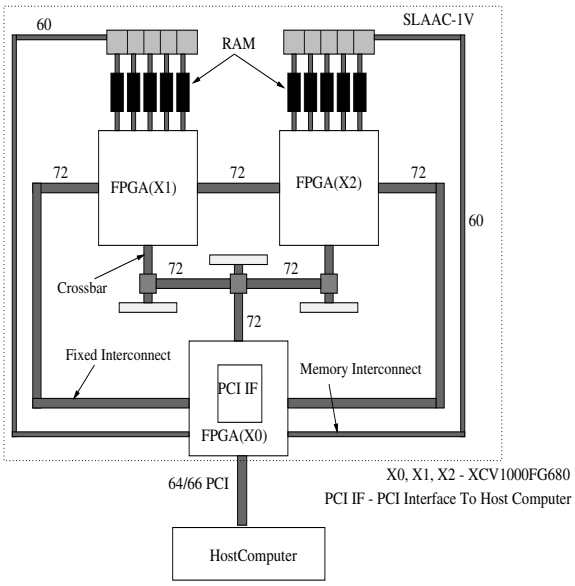

Figure 4. *SLAAC-1V Architecture*

and input data multiplexers. Input data multiplexer is required to select the correct input to pipeline stages. The data is selected between the memory and the intermediate data. Intermediate transfer occurs only when the pipeline folds. Consider an example where there are 6 pipeline stages (virtual stages) and the device can fit only 3 pipeline stages (physical stages) at a time. When stage 4 is executing in the first column of the device, its inputs are the outputs of stage 3. Hence, The intermediate data of stage 3 is given as one of the inputs to the data multiplexer. The data multiplexer selects between the two inputs depending on the select signal which is generated by the state machine. The data controller is generated automatically in VHDL by the controller generator. It is further simulated to verify its functionality, synthesized, placed and routed. Finally, the bitstream for the data controller is generated.

4.6 Host Controller Generation

Host controller dynamically reconfigures the pipeline stages. The controller first initializes the input memory with the input data, after which the pipeline stages are reconfigured one after the other at every clock cycle. While reconfiguring the pipeline stage a single clock step is given to the FPGA so that the previously configured pipeline stages can execute. The lower bound on clock period (which is the throughput of the design) can be calculated by taking the greater among the largest time taken to:

- *execute* any of the pipeline stages
- *configure* any of the pipeline stages

The host controller uses the APIs [2] provided by the SLAAC board to reconfigure pipeline stages, issue clock steps, initialize memories and read back from them. The host controller is generated automatically by the controller generator and the design is executed by executing the host controller.

Design	Physical Stages	Virtual Designs	Partial Designs
FFT	2	2	2
FIR filter	4	4	4
Bubble Sorter	3	3	3
DCT 1D	7	7	7
Elliptic Filter	6	3	8

Table 1. *Number of Physical Stages and Number of Virtual Stages for Examples*

Design	Throughput with VP (outputs/sec)	Throughput without VP (output/sec)	Speed Up
FIR	47M	22.4M	2.098
Bubble	33M	18.653M	1.769
DCT	41M	32.8M	1.25
Elliptic	14.49	2.604	5.56

Table 2. *Throughput Comparison Using Virtual Pipelining (VP)*

5 Results

Several designs were taken through the design flow and were executed on the SLAAC-1V board to validate the framework. In our framework every design is characterized by the number of physical stages (pipeline stages present in the design) and number of virtual stages (the number of stages the device can fit at a time). The physical stages (p) and virtual stages (v) for each design is given in Table 1.

Table 2 shows the gain in throughput due to virtual pipelining. In the table 'M' indicates that the throughput is in the order of Mega output per second. From the table it is inferred that the throughput is only dependent on execution time when the number of physical stages (p) is equal to or more than number of virtual stages (v). When p is less than v, pipeline folds, because of which throughput not only depends on execution time of stages but also on reconfiguration time (refer Section 4.6). For all the above examples except the elliptical filter, the number of physical stages was greater than the number of virtual stages and hence the throughput was entirely dependent on the execution time. For the elliptical filter, reconfiguration was needed during every clock cycle. Also, as reconfiguration time (in the order of milli seconds) is much greater than the execution time

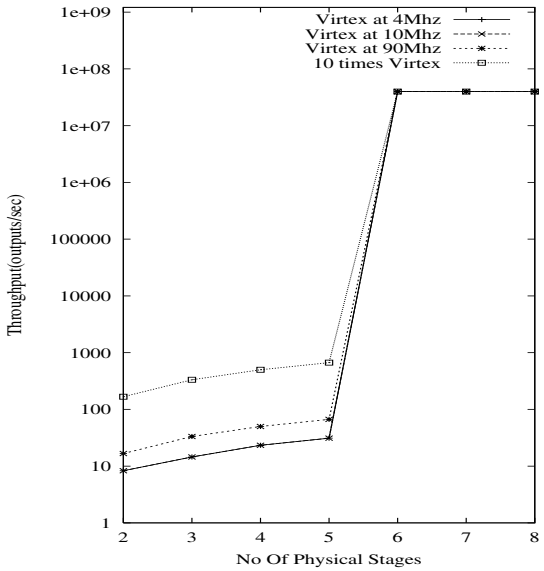

Figure 5. *Throughput vs Number of Physical Stages for Various Configuration time*

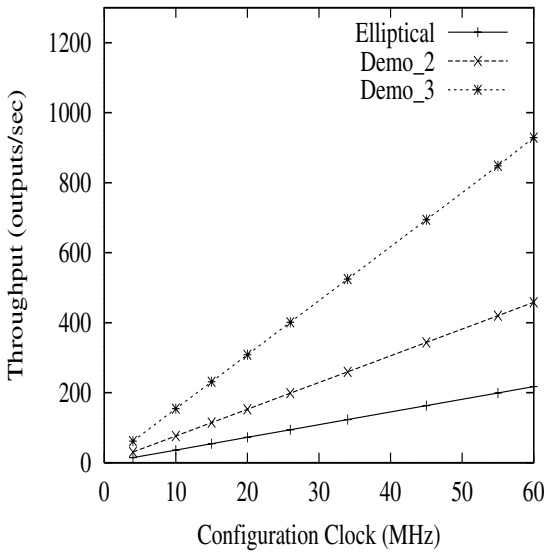

Figure 6. *Throughput vs Configuration Clock Frequency*

for the elliptic filter, the throughput is very low.

5.1 Variation of Throughput

The number of physical stages was varied to study the variation in throughput for various designs (with various *v*). Study shows that throughput increases with the increasing *p* until *p* becomes equal to *v* . When the number of physical stages is less than the number of virtual stages, reconfiguration occurs during every clock cycle. Thus throughput is dependent on the reconfiguration time and the number of physical stages. Once the number of physical stages becomes equal to or greater than the number of virtual stages, there is no more reconfiguration needed, as all stages fit on the device at the same time. Figure 8 shows this variation in a semi-log plot with the number of physical stages on the x-axis and the throughput on y-axis for various examples (or virtual stages).

As throughput is dependent on reconfiguration time, throughput can be increased by decreasing the reconfiguration time. As the reconfiguration time is linearly dependent on the configuration clock frequency, the throughput increase linearly with the configuration clock frequency. Figure 6 shows the variation of throughput with configuration frequency for various examples. The increase in only for the designs when $p < v$, as only theses case needs reconfiguration. For the case when $p > v$, the throughput is independent of reconfiguration time.

The main bottleneck for the low throughput is due to high reconfiguration time. The current technology offers reconfiguration time of the order of milli seconds. With the technology changing rapidly, the reconfiguration time is bound to decrease. The Figure 5 shows the variation of throughput for a elliptical filter with 6 virtual stages. To predict the behavior of throughput various reconfiguration times were assumed. The plot shows that, when the reconfiguration time decreases, the curve shifts higher but then converges to the same constant value.

As discussed in Section 4.6 throughput is dependent on (1) execution time and the (2) reconfiguration time of pipeline stages. To study how these two factor effect the performance (throughput) we plotted the variation of throughput with design clock frequency (which is $\frac{1}{executiontime}$) for various reconfiguration times. The graph in Figure 7 shows that, as the reconfiguration time decreases, the throughput becomes increasingly dependent on the design clock frequency. The throughput is dependent on the reconfiguration time until the reconfiguration time is less than the execution time, later it is dependent only on the execution time.

6 Conclusions

We have presented a framework and design flow to improve throughput of designs using Virtual Pipelining. Targeting pipelined designs on FPGAs, have the advantage of parallelizing the operations present in each stage. But, it suffers from the draw back of finite device size and high reconfiguration time. Virtual Pipelining solves both of the problems, by virtualizing hardware and by using partial reconfiguration instead of full reconfiguration. Using virtual pipelining it is potentially possible to eliminate reconfigu-

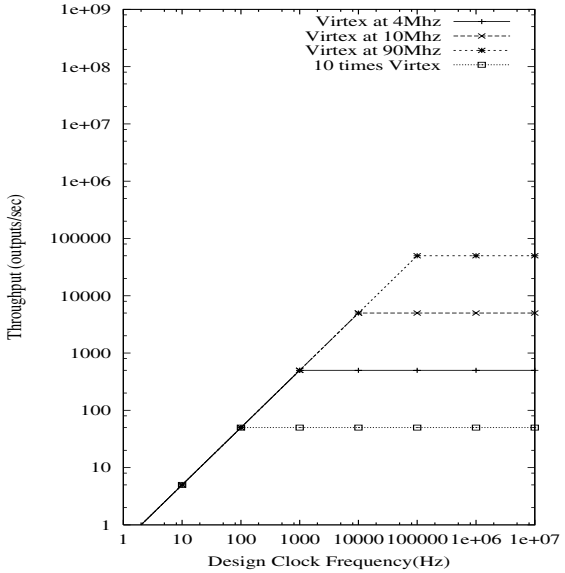

Figure 7. *Throughput vs Design Clock Frequency for Various Configuration time*

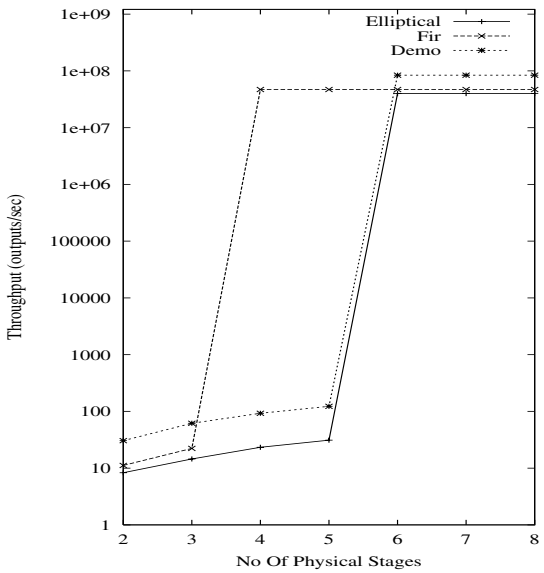

Figure 8. *Throughput vs Number of Physical Stages for Various Virtual Stages*

ration time (using devices with small reconfiguration time).

Central to the design flow is the partial design generator, which takes input specification and splits the design into a number of partial design in JHDL, that has features which enables to structure the layout of the design. In order to aid generation of partial bitstreams, the design is structured so that the components in the design are placed column-wise on the chip. A library of components in JHDL was developed to support a wide range of commonly used operations. We also developed a design flow for Virtex based FPGA board. The design flow was verified on SLAAC1-V board. In order for the synchronization between the host and the reconfigurable board, we presented a host controller that dynamically reconfigures pipeline stages. Data controller which controls the flow of data to and from the virtually pipelined design was also discussed.

The increase in throughput with an increase in the number of physical stages is verified. Also because of excessive reconfiguration time of the device, there is up-to a factor of 5 improvement in throughput using virtual pipelining. When there is no folding of pipeline, the throughput is only dependent on the clock period. As the bottleneck for the low throughput is due to high reconfiguration time, we found that by reducing the reconfiguration time we can greatly improve the throughput.

References

[1] Java hardware description language, release 0.2.

[2] P. Bellows. Slaac-1v reference manual, release 0.3.1, october 2000.

[3] P. Bellows and B. Hutchings. Jhdl - an hdl for reconfigurable systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, 1998.

[4] S. Goldstein, H. Schmit, M. Moe, M. Bidiu, S. Cadambi, R. Taylor, and R. Laufer. Piperench : A coprocessor for streaming multimedia acceleration. In *26th Annual International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.

[5] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[6] H. Schmidt. Incremental reconfiguration for pipelined applications. In *IEEE Symposium on FPGA for Custom Computing Machines*, 1997.

[7] Xilinx. Virtex 2.5 field programmable gate arrays, reference manual, 1999.

[8] Xilinx. Jbits reference manual, 2000.