Hardware synthesis for reconfigurable heterogeneous pipelined accelerators.

Citation for published version (APA):

Jozwiak, L., & Douglas, A. U. (2008). Hardware synthesis for reconfigurable heterogeneous pipelined accelerators. In *Fifth International Conference on Information Technology: New Generations, 2008. ITNG 2008, 7-9 April 2008, Las Vegas, Nevada* (pp. 1123-1130). IEEE Computer Society.
<https://doi.org/10.1109/ITNG.2008.65>

DOI:

[10.1109/ITNG.2008.65](https://doi.org/10.1109/ITNG.2008.65)

Document status and date:

Published: 01/01/2008

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Hardware Synthesis for Reconfigurable Heterogeneous Pipelined Accelerators

Lech Jóźwiak and Alexander Douglas
 Eindhoven University of Technology, The Netherlands
 L.Jozwiak@tue.nl

Abstract

This paper discusses a method of hardware synthesis for re-configurable heterogeneous pipelined accelerators and corresponding EDA-tool that we developed. To evaluate the method and tool, we performed experiments using several representative image and signal processing cases. The experiments showed that our tool is able to automatically construct an optimized hardware that favorably compares to the hardware constructed by skilled human designers, but the tool does it several orders of magnitude faster than a human designer.

1. Introduction

The unique potential of re-configurable heterogeneous systems [1]-[6] and the recent availability of the SoC and platform FPGA technologies that enable efficient implementation of such systems cause that these systems go mainstream both in the embedded system area and supercomputing. What restrains their wide proliferation is lack of adequate development methodologies and electronic design automation (EDA) tools. This paper addresses the problem of hardware synthesis for the re-configurable heterogeneous pipelined accelerators, and discusses a hardware synthesis method and corresponding EDA-tool that we developed for this aim. Using the tool and several representative real-life image and signal processing cases, we performed a series of synthesis experiments to evaluate the method and tool. For each of the cases our tool automatically constructed a correct parallel pipelined hardware implementation expressed in VHDL that favorably compares to the known high-quality hardware implementations. Some of the implementations synthesized by our tool are presented and discussed in the paper.

2. Main issues of accelerator synthesis

In the actual practical designs, the application computation processes are originally specified in high-level modeling or programming languages (e.g. Matlab, C, C++, etc.). In the synthesis chain considered here, the original computation process specification is first translated into a hierarchical conditional dependency graph (HCDG) representation [7][8] being a sort of a conditional data-flow graph. An elementary accelerator corresponding to a (nested) loop body or another partial computation selected for acceleration will be referred to as a cell. A complex heterogeneous accelerator is composed of a number of (different) cells. The input to the accelerator hardware synthesis chain consists of a HCDG representing a given partial computation process (e.g. a loop body) that has to be accelerated, the specification of the cell resources available to implement the computation process, the optimization objectives, and the relevant trade-off information. The hardware synthesis chain involves the high-level, RTL-level, circuit-level, and physical-level hardware synthesis. This paper considers the high-level and RTL-level accelerator hardware synthesis.

During the high-level synthesis, the HCDG representing a given partial computation process that has to be accelerated is first assigned and scheduled, when observing the resource constraints, optimization objectives and trade-off information. This can be performed using various methods, as for instance the constraint programming based methods discussed in [7][8]. Fig. 1 shows the initial C code and equivalent HCDG representation of an example partial computation representing a certain kind of matrix processing typical to signal processing applications. In Fig. 1, different colors are used to indicate different types of HCDG vertices and edges, in particular:

- **magenta** lines between two vertices indicate *control flows*;

Fig. 1 Application C code and corresponding HCDG

- **blue** edges indicate data flows were data is going from a vertex that produces data at time t_1 to a vertex that produces data or a control at time t_2 , where $t_2 > t_1$;
- **red** edges indicate data flows were data is going from a vertex that produces data at time t_1 to a vertex that produces data or a control at a time t_2 , where $t_2 < t_1$ (for example, the last line in the loop body: $e_{new} = e_{old} + d$); the edges indicated by a red color are referred to as **dependency edges**.

The hierarchical conditional dependency graph as shown in Fig. 1 represents only the loop body executed by the accelerator. The loop instruction itself is handled by the CPU-centric processor or another controller controlling the accelerator. In Fig. 1 a specific HCDG node is distinguished referred to as **root node** that is the highest-ranking node in the guard hierarchy. The situation where this guard evaluates to true represents a function call to the computation that the graph represents. The upper part of Fig. 1 shows the example resource constraints for the HCDG from Fig. 1 which make explicit the resources of each particular type. In this example, there are four inputs, one adder, one multiplier etc. Since both adder and multiplier are present, parallel processing is possible to

Fig. 2 Example of resource constraints (upper part) and schedule (lower part)

speedup the computations. The HCDG is assigned and scheduled when observing the resource constraints. The schedule resulting is represented in the lower part of Fig. 2. The scheduling information in Fig. 2 indicates that a three-stage pipeline can be synthesized for the corresponding accelerator to further speedup the execution.

The HCDG representing the cell's computation process, as well as its resource constraints and schedule that limit the implementation freedom, together constitute the input information for the RTL-level accelerator hardware synthesis process on which the remaining part of this paper is focused.

The main concepts on which the synthesis of the accelerator hardware is based are the following:

- synthesis of the application-specific processing units with tailored processing and data granularity;
- parallelism exploitation for execution of a particular computation instance due to availability of multiple application-specific operational resources working in parallel;
- parallelism exploitation for execution of several different computation instances at the same time due to pipelining.

The main decisions related to the usage of the (application-specific) processing units and parallelism exploitation in the execution of a particular partial computation process are taken during the high-level allocation and scheduling processes. However, these decisions have to be then adequately implemented and additional decisions have to be made during the RTL-level hardware synthesis to actually result in high-speed, low-area and low-power accelerators. In particular, the high-level synthesis results correspond to an application-specific functional unit (data-path) enabling parallel execution of a particular computation instance. For the example computation specification from Fig. 1, the corresponding parallel application-specific data-path is represented in Fig. 3. However, the decisions regarding the pipelined execution of different computation instances are only partly made during the high-level synthesis. One of the main results of the high-level synthesis is the schedule that indicates that pipelining with a particular number of pipeline stages can be realized in the corresponding accelerator to further speedup the computation execution comparing to the non-pipelined parallel realization. However, the results of the high-level synthesis do not specify any implementation of the possible pipelining, while there are several different possibilities to extend the basic parallel data-path (as e.g. in Fig. 3) into a corresponding pipelined data-path. We analyzed and evaluated several different pipelining architectures. Two of them: the pipelined data-path based on parallel

Fig. 3 Basic parallel data-path (without pipelining) for the computation example from Fig. 1

registers and pipelined data-path based on serial registers will be discussed in more detail below.

The pipelined data-path based on parallel registers is obtained from the basic parallel data-path through:

- replication of each register of the basic data-path so that the number copies is equal to the number pipeline stages in the schedule,
- introduction of a multiplexer on the output of each replicated register set, and
- appropriate interconnection of the registers with multiplexers and with the resources of the original basic data-path.

For instance, the pipelined data-path based on parallel registers obtained from the basic parallel data-path shown in Fig. 3 is given in Fig. 4.

The number of states of the controller necessary to control the pipelined data-path based on parallel registers is twice as high as the number of the pipeline stages. The first state set containing the number of states equal to the number of pipeline stages is required to control the pipeline start-up, and the second equally large set of states to control the regular pipeline operation in its steady state. Every state of the controller involves as many cycles as there are cycles in the schedule.

The pipelined data-path architecture based on parallel registers has several disadvantages:

- a large number of otherwise unnecessary registers;
- a large number of additional multiplexers;
- many extra control signals required (all the registers and multiplexers have to be controlled by

the controller: many interconnections between the data path and the controller);

- many extra data-path connections required (all the registers of each different register set have to be connected in parallel to the same input and their outputs to the corresponding multiplexer);
- a relatively high number of states in the corresponding controller.

These disadvantages have dramatic consequences for the further implementation of the pipelined cell hardware in the FPGA or SoC technologies: many extra hardware resources will be needed, and the placement and routing will be complicated. Therefore, in our automatic accelerator hardware synthesis method, we do not use the pipelined data-path architecture based on parallel registers, or any similar architecture.

An alternative is the pipeline scheme based on serial registers. In this scheme, some extra serially connected registers are used to make data available to the subsequent pipeline stages. In particular, a single register is used to make data available to the next pipeline stage. For example, consider the *o* input in Fig. 1 and Fig. 3. The data of the *o* input produced in pipeline stage 0, is consumed by node *d* in pipeline stage 2. This means that two registers in series are needed to transport the data from stage 0 to stage 2. The pipelined cell architecture obtained for the same basic parallel data-path from Fig. 3 and the same schedule from Fig. 2 when using the serial register chains is represented in Fig. 5. The pipelined cell architecture exploiting the serial register chains is significantly simpler than the architecture based on parallel registers. To obtain the architecture as shown in Fig. 5 the guard information is also used. Because guard *g2* is the inverse of guard *g1*, the two guards are mutually exclusive. This means that operations executed under control of guard *g1* and guard *g2* can share registers to even further reduce the amount of registers required by the data path. Take for instance node *a* and node *b* of the HCDG in Fig. 1. Since their controlling guards are mutually exclusive and the nodes are scheduled at the same point in time, only one of the operations as specified by node *a* and node *b* will be performed. This means that the registers for the results of these two operations can be shared. The pipelined cell architecture based on the serial registers (as in Fig. 5) does not have any of the disadvantages as identified for the architecture based on the parallel registers. In particular, the number of additional registers required to implement pipelining is greatly reduced, the additional multiplexers due to parallel registers as avoided, the controller is roughly twice smaller, and much fewer interconnections both in the data-path and between the controller and data-path are

Fig. 4 Pipelined data-path based on parallel registers

needed. In consequence, the resulting additional hardware to implement pipelining and the power dissipation are greatly reduced, when also positively influencing the circuit speed due to a smaller area and shorter interconnects. Consequently, we use the pipelined architecture based on serial registers in our method.

The above is only one of a larger set of general static optimization decisions (i.e. decisions independent on a particular computation process to implement, particular constraints and objectives) that had to be taken to construct an adequate hardware synthesis method and corresponding tool. Also, an apparatus for the more specific dynamic optimization decisions, that are taken by the synthesis tool during its operation and depend on a particular computation process, constraints and objectives has been developed and built in our synthesis method and corresponding tool. Discussion of all these decisions is out of the scope of this brief conference paper, and therefore, we only focus on explanation of the main issues.

One of the main issues is the cell's processing unit interfacing with the rest of the system. As it can be observed in Fig. 5, the cell's processing unit – both the cell's data-path and controller – require several interconnections to interface with the rest of the system. This interface is fully defined by the inputs and outputs of a particular computation process to be

Fig. 5 Pipelined data-path based on serial registers

implemented and decisions that are explained below. We pre-designed a generic VHDL interface definition in the entity description of the architecture that is appropriately instantiated for each particular computation process realized with a particular processing unit. For example, for the cell's processing unit in Fig. 5, the corresponding VHDL interface description generated by our tool is as represented in Fig. 6. Regarding the input and output signals, the resource constraints specify that all input and output data is 16 bit wide, and consequently, the corresponding 16 bit busses are instantiated in the VHDL description. In addition to the input and output signals the entity defines three specific signals that are required by the rest of the system to adequately control the cell's hardware operation:

- clock signal **clk** needed because the cell's hardware is synchronous;
- reset signal **rst** needed to ensure that the controller starts in the correct initial state;
- root signal **root** needed to activate/stop the cell operation by the external world – a particular cell does not necessarily to be active continuously: the **root** input can be used to halt all the cell's operations by making it false ('0'), and by doing so, to stall the pipeline, until the **root** is true ('1') again.

Note that output **e** is not declared as an actual output in the entity description. This is due to the fact that **e** is

```

entity cell is
  port ( clk      : in std_logic;
        rst      : in std_logic;
        root     : in std_logic;
        k        : in signed (15 downto 0);
        l        : in signed (15 downto 0);
        m        : in signed (15 downto 0);
        o        : in signed (15 downto 0);
        e        : buffer signed (15 downto 0));
end cell;

```

Fig. 6 VHDL code of the cell's processing unit interface

also needed internally (Fig. 5). By declaring *e* as buffered, its value can also be accessed for further internal processing. The buffer port declaration differs from the inout declaration, because for a port that is declared inout, the value of the bus is read back, while for the buffer declaration the value of the driver is read back. Alternatively, an alias could have been used. By doing this, port *e* could be declared as an output, and the alias could be used for reading back the *e* value. We did not use this possible construction, because the alias construction of VHDL is not supported by several circuit synthesis tools.

3. Hardware synthesis method for heterogeneous accelerators

The **main steps of the accelerator cell hardware synthesis method** are as follows:

1. **Define the interface of the cell's processing unit:**
The interface always includes the root signal, reset signal and a clock signal. Additional input and output signals are added to the interface dependent on the HCDG, the resource constraints and the schedule of a particular computation that has to be implemented in the cell. These additional signals include all inputs and outputs as specified by the resource constraints with the names as defined in the HCDG (for the HCDG in Fig. 1 and constraints in Fig. 2, the additional signals are the *k*, *l*, *m* and *o* inputs). Moreover, based on the HCDG and the schedule, all data leaves should be identified and added as outputs to the cell interface (for the HCDG in Fig. 1. and schedule in Fig. 2, *e* should be added as output to the cell interface).
2. **Construct an optimized cell's pipelined parallel data-path:**
 - 2.1 **Determine operand multiplexing for operational units through considering construction of the**

basic parallel cell's data-path for the given HCDG, resource constraints and schedule: An available resource as indicated by the resource constraints might be used to execute the operation as indicated by multiple vertices in the HCDG. If this is the case, then a multiplexer is needed to select one of the possible operands for the operational unit's input. For example, the operations specified by vertices *a*, *b* and *d* of the HCDG in Fig. 1 are all executed on the same multiplier, because there is only one multiplier resource available (Fig. 2). In consequence, the operand B of the multiplier is connected to a multiplexer capable of selecting one of the following signals: *l*, *m*, *mux output* (Fig. 3).

- 2.2 **Determine serial register chains:** To implement pipelining serial register chains are used. For each vertex in a given HCDG (except of the root vertex), a register chain must be determined. As explained above, an appropriate number of registers in the register chain is required, if the output of a vertex (data or control) is directed towards a vertex in a higher pipeline stage according to the schedule. The number of registers in a chain is equal to [largest pipeline stage (data or control) consumer vertex] – [pipeline stage (data or control) producing vertex]. For example, in the HCDG (Fig. 1) the vertex *o* produces data in pipeline stage 0 (iter=2 in Fig. 2). The largest pipeline stage of the consumer, vertex *d*, is pipeline stage 2 (iter=0 in Fig. 2). Therefore, the register chain behind input *o* requires 2-0=2 registers (Fig. 5).
- 2.3 **Minimize the sets of registers after the operational units:** The serial register chains that have been determined in the previous step may contain some redundant registers. If guards, other than the root guard, are present in a HCDG then some of the guards may be mutually exclusive. The mutually exclusive guards can be used to discover the register redundancy. Registers are considered redundant if they are connected to the same operational unit, and are used in the same pipeline stage. For example, guard *g1* and *g2* in the HCDG (Fig. 1) are exclusive. Vertices *a* and *b* both specify an operation executed on the same operational unit and in the same pipeline stage. This means that either *a* or *b* is computed, but not both at the same time, and indicates redundancy in the register chains corresponding to *a* and *b*. The registers of both chains that provide information for the same pipeline stages can therefore be joined (e.g. resulting in *b_or_a* register in Fig. 5).

- 2.4 Minimize the sets of registers after guard calculations:** Similarly to the registers after the operational units, the registers holding the guard information can also be redundant. If mutually exclusive guards are present in the HCDG, the longest register chain of the guard is kept, while the shorter guard register chain is removed and replaced by the exclusive guard expression (e.g. in Fig. 7 were guard p and q are mutually exclusive).
- 3. Construct the controller and its interconnections with the data-path:** The final step in the cell's processing unit construction is to construct the controller that controls the cell's data-path, and specifically its pipelined processing as defined by the schedule, through controlling the multiplexers and registers as determined in the previous steps and described above.
- 4. Instantiate the cell's memory and communication module:** Finally, the pre-defined cell's memory and communication module are instantiated in a straightforward way to satisfy the memory and communication needs of a particular cell.

We implemented the above discussed RTL-level accelerator hardware synthesis method in the form of an automatic accelerator cell generator. The main interface of the cell generator is its programming interface that consists of several public functions of the generator class that can be used to transfer data to and from the generator, and to start the cell generation process. Some of the functions serve to transfer the generator's input information consisting of the HCDG, resource constraints, schedule, and additional information that is necessary for the adequate cell implementation, as for instance, the word size of the cell operators, active clock edge, active reset level, etc. In result of running the generator with a particular set

Fig. 7 Register minimization after guard calculation

Fig. 8 Input information for matrix vector multiplication

of its input data the generator constructs an optimized pipelined parallel cell implementation at the RTL-level, and generates as its output the corresponding cell's VHDL specification saved in a (.vhd) file, and a report file (.rpt) documenting all the synthesis steps and decisions made in each step.

4. Experimental results

Using the accelerator cell hardware generator, we performed a series of synthesis experiments to test it,

Fig. 9 Hardware for the matrix vector multiplication

and to evaluate our accelerator hardware synthesis method. In the experiments several characteristic image and signal processing computation specifications were used, together with the corresponding resource constraints and schedules, for which their high-quality hardware implementations were known. The synthesis results from our accelerator cell generator were compared against the known implementations. For each of the test cases, our cell generator constructed a correct parallel pipelined hardware implementation that was equally good or better than the known high-quality hardware implementation, and was expressed in a correct synthesizable VHDL code. To verify if the generated hardware functions correctly, its VHDL code has been compiled and extensively simulated. Some of the computation specifications and their corresponding cells constructed by our generator are presented here.

Matrix vector multiplication often used in various signal processing applications can be denoted for a 2x2 matrix as follows:

$$\begin{pmatrix} a1 & a2 \\ b1 & b2 \end{pmatrix} \begin{pmatrix} d1 \\ d2 \end{pmatrix} = \begin{pmatrix} a1d1 + a2d2 \\ b1d1 + b2d2 \end{pmatrix} = \begin{pmatrix} r1 \\ r2 \end{pmatrix}$$

In Fig. 8, the input information for the cell generator is shown. The resource constraints decide that a single rx : $r1$ or $r2$ can be determined in two consecutive cycles. To implement it, two multipliers and a single adder are needed. For this input information our cell generator synthesized the corresponding hardware as represented in Fig. 9.

Fig. 10 Input information of the pixel processing

Fig. 11 Hardware for the pixel processing function

Pixel processing function: Fig. 10 shows the original C code of a pixel processing function typical to image processing applications and its corresponding hierarchical conditional dependency graph with the resource constraints and scheduling information. For this input information, our cell generator synthesized the corresponding hardware as represented in Fig. 11.

Butterfly from FFT: The cell hardware shown in Fig. 12 for the butterfly of the Fast Fourier Transform (FFT) was synthesized according to the input information given in Fig. 14.

5. Conclusion

In this paper, we discussed several issues of hardware synthesis for the re-configurable heterogeneous accelerators, as well as an accelerator hardware synthesis method and corresponding EDA-tool that we developed. Using the tool, we performed a series of synthesis experiments to evaluate the method and tool. In the experiments, several representative real-life image and signal processing cases were processed. For each of the cases, our tool constructed a correct parallel pipelined hardware implementation expressed in VHDL that favorably compared to the known high-quality hardware implementations. Some of the implementations synthesized by our tool are presented in the paper. The tool is able to automatically construct a highly optimized hardware for the typical image and signal processing computations that

Fig. 12 Input information of the FFT's butterfly

favorably compares to the hardware constructed by skilled human designers, but the tool does it several orders of magnitude faster than a human designer. The experimental results confirm the adequacy of the method and tool.

6. References

- [1] S. Choi, R. Scrofano, V. K. Prasanna, and J. W. Jang: Energy-efficient signal processing using FPGAs, Proc. Int. Symp. on Field Programmable Gate Arrays, Feb. 2003, pp. 225-234.
- [2] R. Lysecky and F. Vahid: A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning, in Proc. Design, Automation & Test in Europe Conf., Mar. 2005.
- [3] M.A. Perkowski, L. Jóźwiak, D. Foote, Q. Chen, A. Al-

Rabadi: Learning Hardware Using Multiple-Valued Logic, IEEE Micro, Vol.2, No 3, May/June 2002, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 41-51.

- [4] R. Scrofano, S. Choi, and V. K. Prasanna, .Energy efficiency of FPGAs and programmable processors for matrix multiplication, Proc. Int. Conf. on Field Programmable Technology, Dec. 2002.
- [5] G. Stitt and F. Vahid: The energy advantages of microprocessor platforms with on-chip configurable logic, Proc. of Design & Test of Computers, Dec. 2002, pp. 36-43.
- [6] C. Wolinski, M. Gokhale and K. McCabe: Polymorphous -based Systems: Model, Tools, Applications, Journal of Systems Architecture, Elsevier Science, Amsterdam, The Netherlands, Vol. 49, No 4-6, September 2003, pp. 143-154.
- [7] C. Wolinski and K. Kuchcinski: Global Approach to Assignment and Scheduling of Complex Behaviours Based on HCDG and Constraint Programming, Journal of Systems Architecture, Elsevier Science, Amsterdam, The Netherlands, Vol. 49, No 4-6, September 2003, pp. 489-503.
- [8] C. Wolinski and K. Kuchcinski: A Constraint Programming Approach to Cell Synthesis, DSD'2005 – 8th Euromicro Conference on Digital System Design, August 30st – September 3rd, 2005, Porto, Portugal, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 356-363.

Acknowledgement: The authors are indebted to C. Wolinski for his kind collaboration and valuable remarks.

Fig. 14 Hardware synthesized for the FFT's butterfly