Design and Implementation of Reconfigurable Tasks with Minimum Reconfiguration Overhead

Markus Rullmann and Renate Merker
Dresden University of Technology, Germany
Circuits and Systems Laboratory
{rullmann, merker}@iee1.et.tu-dresden.de

Abstract: Dynamic reconfiguration in today's architectures is associated with high reconfiguration overhead. In this paper, we propose a method for reducing such overhead and demonstrate its application in a real world example. A two step approach is presented here: reconfigurable tasks are designed such that they have a very similar structure; and an automated matching tool is used to identify the structural similarities in tasks. An FPGA implementation flow is presented that takes account of this information. A complete hardware/software integration of the tasks is described. Performance results verify the performance gained with the reconfigurable hardware accelerators in a real world system.

1 Introduction

Reconfigurable computing has received an ever increasing interest in the research community in recent years. The associated problems cover all aspects of digital system design, but a major driver is the potential in flexibility, performance and power consumption of such systems. A disadvantage is still the amount of device programming data required to reconfigure devices at runtime. The configuration data needs to be stored in the system memory and transferred to the programming interface of the reconfigurable devices. A large amount of configuration data has a negative impact on system cost, reconfiguration time and power dissipation.

Many groups consider the tasks as being implementation independent [WP03, ABF+05, UHGB04]. Tasks are implemented as independent modules that can be loaded at runtime. The fact that the implementation of many algorithms can include a large degree of structural similarities is usually neglected. There are several approaches to identify structural similarities in tasks, e.g. [RWT05, RV01, AS05, MBdSA05]. However, we propose a unique two step approach: At first, the tasks are designed to be as similar as possible and secondly, the implementation tools are advised to reuse part of the configuration information between tasks. This is supported by our matching tool which identifies structural similarities automatically using a systematic approach. Our aim is to reduce the data required for partial reconfiguration.

The rest of this paper is structured as follows: At first we briefly introduce the concept of *hardware matching* in Sect. 2. In Sect. 3 we present two algorithms that are implemented

as reconfigurable tasks. The implementation strategy is described and the similarity between both tasks is illustrated on a high level. The tasks were integrated into an embedded, reconfigurable hardware/software system and performance measures are given. Finally, in Sect. 4 we evaluate a design flow that integrates the hardware matching into a vendor's standard flow and show that it indeed reduced the required number of configuration bits.

2 Hardware Matching

Generally tasks are implemented by the placement and routing tools independent of each other. Hence the tools generate different configurations for similarly structured logic in each of the tasks. When such tasks are reconfigured, a large amount of configuration data is used to implement the same similarly structured logic on different physical resources. We call this *redundant configuration.* The overhead associated to reconfigurable systems can be reduced by eliminating the redundant configuration.

In our approach we search for structural similarity of the different tasks, i.e. for such parts that can be both mapped to the same resource types and placed and routed using the same physical resources. An automated tool has been developed that identifies similar structures in two different tasks. The input to the tool are netlists obtained from any HDL synthesis tool. The netlists are internally converted into an equivalent graph representation. We described the Maximum Edge Matching Problem for such graphs in [RM06] and proposed two possible algorithms to solve it.

The solution to the Maximum Edge Matching Problem defines the logic resources that must be placed on the same physical resources in order to achieve a maximum routing similarity between two tasks. The solution also provides a measure, the matching weight, that states the number of routes that should use the same routing configuration. Ideally, the implementation tools take advantage of these information and map, place and route the given tasks accordingly. The reconfiguration will then consist of a reconfiguration of all non-matching routes and the difference in the logic resources, e.g. configuration of look up tables (LUT) etc.

3 Application Example

In our case study, we implement two functions of the H.264 Video Encoding Standard as hardware tasks in a reconfigurable embedded system. Our hardware tasks are able to support two of the most computation intensive functions in the video encoder. Task 1 will implement the integer transform as it is defined by the standard and task 2 will support the motion estimation engine as a hardware accelerator.

The implementation of both task is designed as processor array (PA) that consists of a regular structure of identical processing elements (PE).

Task 1: Integer Transform

The integer transform (IT) is used in H.264 to convert the pixel data into a transform domain. The redundancy of the pixel data can be much easier exploited for compression in the transform domain. The standard proposes the integer transform because its properties are similar to the discrete cosine transform which has been used in earlier video compression standards, but its computational complexity is much less [WSBL03]. Furthermore, it allows for perfect reconstruction if no quantization is applied. The formal definition of the IT is:

$$S = H \cdot B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix} \cdot B$$

where B is an data block of 4×4 pixels. An entire frame is transformed on a block by block basis.

Task 2: Motion Estimation

Our motion estimation kernel is based on blocks of 4×4 Pixels. It computes the sum of absolute differences (SAD) operation for a block. A mathematical formulation is given below:

$$S = \sum_{x=0}^{3} \sum_{y=0}^{3} \mid B_1(x,y) - B_2(x,y) \mid .$$

Embedded in a motion estimation for video encoding, the search strategy defines for which data blocks B_1 and B_2 the SAD is computed. In consequence, the motion vector is derived from the blocks with minimum SAD. The minimum SAD is selected by the software and is not contained in the PA.

Algorithm Partitioning

Both tasks were divided into a hardware and software part. The hardware part performs the arithmetic operations for one partition of the algorithm with data stored in a local RAM. The software part organizes the execution order between partitions by means of data transfers to the local memory of the task. The hardware part implements 4 parallel PEs for the datapath operations. The required datapath in each PE for Task 1 consists of one simplified multiplication and accumulator and for Task 2 of one subtraction and one accumulator. A sequential hardware execution controller executes a partition of the overall algorithm independently from the main CPU. Logically, this divides the implementation of the regular algorithms in three levels: software control, sequential hardware execution control and parallel hardware execution. The hardware partition has been designed to gain maximum utilization of the local memory [SM04]. This reduces the data transfer and control overhead by the software and thus increases overall the performance.

3.1 Task Architecture Template

Our implementation is based on an architecture template which has been designed to allow a straightforward implementation of tasks based on PAs. The template integrates the PA, the global PA controller and local memory buffers for the PA. A generic task controller implements the system bus interface and a task control, which is identical for all tasks using this template. The template is depicted in Figure 1.

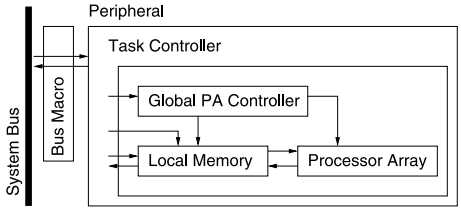


Figure 1: Task Architecture Template

The global PA controller steers the execution of the algorithm on the PA. Therefore it generates the necessary control signals for the PA and also calculates the addresses to control the dataflow between the local memory and the PA. The global PA controller is based on a reconfigurable state machine and several reconfigurable address generators. The configuration of the state machine and the address generators is automatically derived from PA design parameters [KMS03]. A reconfiguration of the control flow can be achieved by reconfiguration of the CLB ROMs without additional overhead in the logic design.

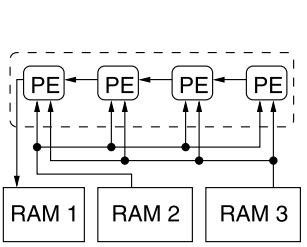
The architecture template features a common task controller with a unified set of registers. The controller supports the following functions: task identification register, start/stop of the task, set the task execution for a defined number of cycles, read the runtime task state, flexible task control via a generic control register and a reset to initialize the task to a known state after reconfiguration. The controller registers can be accessed through the system bus to control the execution of the task.

3.2 Processor Array Design

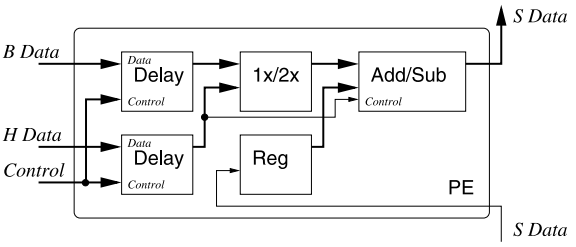
The control signals generated by the state machine of the global PA Controller drive the execution schedule of the PA. The application data is directly transferred from the local memory to the PA. All data delays and pipeline registers are implemented together with the datapath operations in the PA.

In Figure 2(a) the PA for the task implementation is shown along with the structure of the PEs for each task. The PA features the same datapath structure for both tasks. The local RAMs contain the data specific to each task: RAM 1 – S Data, RAM 2 – H, B2 Data and RAM 3 – B and B1 Data. The PA is designed such that the implementation of both tasks have a high structural similarity. Figure 2(b, c) shows the internal structure

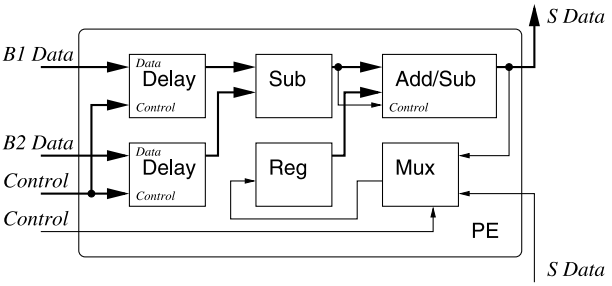
of the PE. Most datapath connectivity is the same for both tasks, on the assumption that the simplified multiplication and the subtract operation can be implemented with the same physical resources in the target architecture. This is achieved with a custom design of the simplified multiplier/accumulator datapath. Hence we do not only implement a similar structure of the datapath, but also of the underlaying basic logic resources. It must be pointed out that no additional logic is used to maximize similarities, hence we do not introduce additional overhead here.



(a) Processor Array Datapath Structure



(b) Processing Element for the integer transform



(c) Processing Element for SAD

Figure 2: Architecture of the common processor array (a) and the processing elements (b, c) specific for both tasks. Common datapath connectivity in (b, c) are drawn bold.

3.3 Hardware/Software Integration

The tasks have been integrated into our embedded hardware/software system [WRM05]. This system consists of the embedded PowerPC processor on the FPGA and the CoreConnect based system periphery. The reconfigurable module is connected via an On–chip Peripheral Bus (OPB) slave interface. The CPU is clocked with 300 MHz, the bus speed is 100 MHz. The system is operated under Linux with the necessary drivers for the internal configuration access port (ICAP) of the FPGA and drivers for the tasks. The current implementation provides free resources for one module that can occupy 2×40 configurable logic blocks (CLBs) and ten 18 KBit embedded RAM blocks.

3.4 Algorithm Performance

The algorithm performance has been measured with benchmark software implemented on the embedded Linux OS. The results show a significant improvement over the software–only implementation. The performance data were measured for two image sizes, e.g. 1024×1024 and 256×256 pixels.The results are slightly different because of a different partitioning used for each size. This influences reused data form the local memory and hence performance. The results are shown in Table 1. The hardware accelerated variants gain a significant speedup of 2.5 and 11.6 for the integer transform and the motion estimation, respectively. It could be further improved by attaching the hardware task directly to the Processor Local Bus (PLB) and thus reducing the data transfer bottleneck.

The data transfer bottleneck also explains the different speedups gained. The integer transform task has much less internal data reuse and requires a high I/O bandwidth. In contrast, the motion estimator operates much more on data stored in the local RAM making the data transfers to the CPU less critical.

Image Size		IT		SAD	
		SW	HW	SW	HW
1024^2	Cycles/Block	860	343	56302	4837
	Speedup		2.5		11.6
	#Blocks	65535		61504	
256^2	Cycles/Block	938	431	50464	4646
	Speedup		2.2		10.9
	#Blocks	4096		3136	

Table 1: Algorithm performance measured in the embedded system. Cycles are given in CPU cycles; the #Blocks are the number. of transformed blocks and the number of processed reference blocks in the IT and SAD algorithm respectively.

4 A Matched Implementation Flow

In this case study we demonstrate how the FPGA design flow given by the vendor can be modified in order to incorporate the matching information into the final implementation. The design can be described in any synthesizable hardware description language.

The synthesis results are then analyzed by our matching tool. The tool processes netlists in edif format, because this format is supported by many commercial implementation and synthesis tools. The user can configure the matching tool to optimize runtime and accuracy of the result. He can also supply a priori information about matching instances. This will speed up the matching process but is not vital for the tool. The matching tool transforms the input netlists such that matching instances and connections (nets) have equal names in both tasks. Additionally it outputs textual and statistic information about the matching result.

The transformed netlists can be used directly as input for the implementation tools. The netlists are translated like standard netlists first. Using Xilinx ISE tools, the steps consist of translate, map, place and route. In addition during the map, place and route steps the tools allow the designer to provide an already implemented reference netlist as a guide file. The tools will apply the implementation information from the reference to the actual design based on net names and connectivity – and instance names and types. The result is an implementation that is much more similar to the reference than one that is implemented independently. The benefit of the procedure are reduced reconfiguration costs.

In the example, we used two different implementation flows. The first flow, depicted in Figure 3(a), is the standard flow provided by the FPGA vendor. It is used as a reference to compare the reconfiguration overhead. In our matched implementation flow, the HDL Designs A and B are processed by the matching tool first. HDL Design A is then implemented without a reference netlist. The placed and routed design A is used later in guided map, place and route steps as a reference for the implementation of HDL Design B.

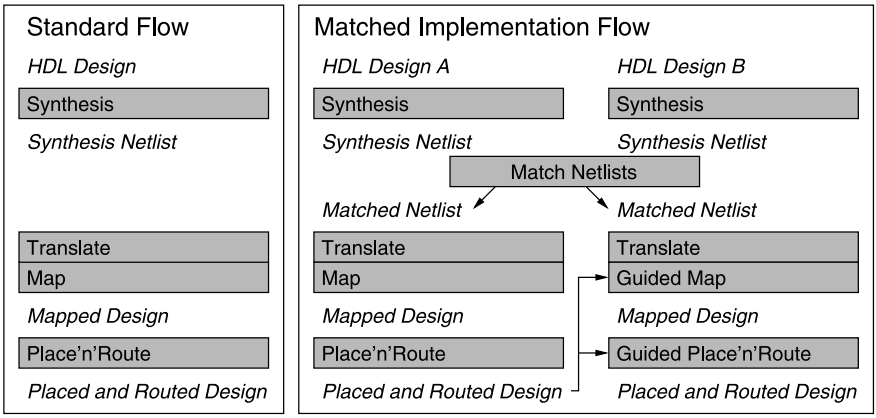


Figure 3: Standard implementation flow (a) and our implementation flow including matching (b).

4.1 Matching Results

In this section we will discuss the matching obtained by our matching tool. The task implementation consists of two parts: the processor array and the control part including the local memory, task controller and the PA controller. The control part is synthesized from a common HDL description for both tasks. However, this part is not static. The PA controller generates a control sequence for both tasks, which is achieved by programming different data into the LUT ROM for the state machine. The synthesiszed netlists for this part differ only in the LUT ROM contents – no matching is needed because instances and nets have already equal names.

The matching tool has been applied to the processor array part in order to identify the struc-

tural similarities in both tasks' implementations. The automated tool is able to identify a large amount of similar structures. We compared the result to a user specified matching. Furthermore we applied the tool to a single PE which gives some indication how the performance of the tool depends on the problem size. The results are summarized in Table 2. It can be seen that the tool identifies apparently all common structures for a single PE, but lacks some matching for a full PA. However, the tool extracts the matching information automatically which is vital in all cases where this information is not known in advance or in automated tool flows. The matching tool does not rely on the self-simliarity of the PA design but rather on the low level structure of the used logic resources.

	Matching Tool	User defined
Processor Array	536	771
Processing Element	223	200

Table 2: Number of matching connections in both tasks extracted by the matching tool compared to a user defined matching. The user defined matching is based on the assumption that logic at the same PE and bit position in the datapath match.

4.2 Experimental Results

We applied the two flows to the two given tasks in order to explore the impact of matching and guided implementation on the total reconfiguration costs. We compare the bitstreams by the number of different configuration frames and the number of different bytes. Configuration frames are given by the granularity of the configuration logic in the FPGA and the difference in bytes provides some hints about the real amount of reconfiguration needed. The benefits in the byte measure can not turned into shorter reconfiguration times due to the limitation of the configuration architecture.

The efficiency of the guide mode has also been evaluated. We analyzed the exact routing of the implemented designs. Our analysis tool determines the number of nets that have the same driver (based on the physical implementation) and the number of connections in these nets that are configured equally in both designs.

We evaluated configurations generated by the standard flow only and the matched implementation flow for both options: Design A= Task 1, Design B= Task 2 and Design A= Task 2, Design B= Task 1. The results are shown in Table 3.

The results clearly demonstrate that the implementations based on the matched implementation flow reduce the configuration difference in bytes by a factor of up to 2. The guide tool clearly takes advantage of the structural similarity identified by our matching tool. The number of common net pins and the number of programmable interconnects (PIP) increase significantly. Still the result is not as impressive as we might have expected. The tool's report files and the routing analysis suggest that the guide mode of the implementation tool is not able to translate all mapping, placement and routing information from the reference design into the new design. Hence an additional penalty in configuration over-

Design A Design B	IT SAD	IT, SAD, Matched	SAD, IT, Matched
#different bytes	14041	9324	6714
#different frames	63	63	62
common net pins	555	1876	1946
common net PIPs	1484	5536	6017

Table 3: Difference in configuration data and routing obtained through various design flow options. For each option, the resulting bitstream difference and common routing configuration is given.

head must be paid, which could be eliminated with *reconfiguration aware* implementation tools. The input designs are optimally prepared as an input to such a tool.

5 Conclusion

In this paper we described a design and implementation method to reduce reconfiguration overhead. It has been shown how the design parameters of two tasks can be chosen to increase the structural similarity of this tasks. In the given examples, this has no impact on the performance of the tasks and no additional implementation overhead is introduced. Our matching tool is able to identify the structural similarities intended by the designer. This enables an automated matched implementation flow. We found that the implementation tools can take advantage of such similarity to reduce the reconfiguration overhead, e.g. the difference between the configuration bitstreams. The results also indicate that there is more potential if the tools would be truly reconfiguration aware. The frame based configuration architecture found in the VirtexIIPro platform poses additional restrictions on our technique.

We have further shown how the reconfigurable tasks can be integrated into an embedded hardware/software system. We compared a software–only to the hardware accelerated implementation and found a significant speedup of 2.5 and 11.6.

References

[ABF+05] A. Ahmadinia, C. Bobda, S. Fekete, T. Haller, A. Linarth, M. Majer, J. Teich, and J. van der Veen. The Erlangen Slot Machine: A Highly Flexible FPGA-Based Reconfigurable Platform. In *In Proceedings of the 2005 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 319–320, April 2005.

[AS05] D Aravind and A Sudarsanam. High Level - Application Analysis Techniques & Architectures - To Explore Design possibilities for Reduced Reconfiguration Area Overheads in FPGAs executing Compute Intensive Applications. In *Parallel and Dis-*

tributed Processing Symposium, 2005. Proceedings. 19th IEEE International, April 2005.

[KMS03] Jürgen Kelber, Renate Merker, and Sebastian Siegel. Systematische Generierung des Steuerflusses von Prozessorarrays. In *Proceedings DASS 2003 and SDA 2003*, pages 49–54, 2003.

[MBdSA05] Nahri Moreano, Edson Borin, Cid de Souza, and Guido Araujo. Efficient Datapath Merging for Partially Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):969–980, July 2005.

[RM06] Markus Rullmann and Renate Merker. Maximum Edge Matching for Reconfigurable Computing. In *To appear in: Proceedings of the 20th IEEE International Conference on Parallel and Distributed Processing Symposium*, April 2006.

[RV01] Daler Rakhmatov and Sarma B. K. Vrudhula. Minimizing Routing Configuration Cost in Dynamically Reconfigurable FPGAs. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 1481–1488, April 2001.

[RWT05] Krishna Prasad Raghuraman, Haibo Wang, and Spyros Tragoudas. A Novel Approach to Minimizing Reconfiguration Cost for LUT-Based FPGAs. In *VLSI Design, 2005. 18th International Conference on*, pages 673–676, January 2005.

[SM04] Sebastian Siegel and Renate Merker. Optimized Data-Reuse in Processor Arrays. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP 2004)*, pages 315–325, September 2004.

[UHGB04] M. Ullmann, M. Hübner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proceedings 18th International Parallel and Distributed Processing Symposium*, pages 135–142, April 2004.

[WP03] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *DATE '03: Proceedings of the conference on Design, automation and test in Europe*, pages 290–295, Washington, DC, USA, March 2003. IEEE Computer Society.

[WRM05] Andreas Weder, Markus Rullmann, and Renate Merker. Ein Linux–basiertes, dynamisch rekonfigurierbares Hardware–Softwaresystem auf Basis der Xilinx ML300 Plattform. In *Dresdner Arbeitstagung Schaltungs- und Systementwurf DASS 2005*, April 2005.

[WSBL03] T. Wiegand, G.J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2003.