

# Design and Validation of Execution Schemes for Dynamically Reconfigurable Architectures

Tobias Oppold, Sven Eisenhardt, Wolfgang Rosenstiel

*Department of Computer Engineering, University of Tuebingen  
Sand 13, 72076 Tuebingen, Germany  
crc@informatik.uni-tuebingen.de*

**Abstract**—Mapping applications onto reconfigurable architectures can be done in many different ways. The features of the target architecture constrain the way an application can be mapped and executed significantly. Execution schemes are generated as an intermediate format in our approach to application mapping and constitute a useful level to compare features of different architectures. In this paper, we present execution schemes that take advantage of fast reconfiguration and results from mapping execution schemes to a commercial architecture.

## I. INTRODUCTION

Traditionally, reconfigurable devices are deployed because of their flexibility to change the application over time. Newly developed architectures can be reconfigured within one clock cycle so that components of a device can be re-used within a single application. We call such architectures *processor-like reconfigurable* architectures.

The reconfiguration keeping pace with the execution yields an additional degree of freedom that we explore in a design environment that is focused on taking advantage of reconfiguration to optimize area, power, and performance.

In our design environment, we evaluate how processor-like reconfiguration can be exploited by a high-level compiler and which architectural resources are needed for an efficient mapping of applications. For an assessment of the benefits and costs imposed by those resources, we developed the CRC model (Configurable Reconfigurable Core) as a general model for processor-like reconfigurable architectures. Instances of the CRC model are synthesized and evaluated at the gate-level using commercial tools.

In previous publications, we have shown how to take advantage of reconfiguration to optimize area [1][2] and power [2] under a given performance constraint. For such purposes we generate execution schemes that describe the execution of an application at the register-transfer (RT) level. For this contribution, we mapped execution schemes to the DRP (Dynamically Reconfigurable Processor) architecture developed by NEC [3] to show the applicability of our approach to a commercial architecture and to point out differences between our CRC model and the DRP architecture.

In contrast to many other reconfigurable architectures, the C compiler for the DRP partitions an application into several configurations automatically so that we were able to generate the execution schemes solely by setting compiler constraints

and by transformations of the source code. NEC developed this compiler based on their hardware synthesis tool Cyber [4].

Before we describe the experiments and discuss the results, the CRC model and details on the DRP architecture are presented in the following section. Section III describes the techniques that we use for generating execution schemes and points to related work.

## II. ARCHITECTURE MODEL

The CRC model was developed to represent a wide range of processor-like reconfigurable architectures. In its most general specification, only a few features are defined: It consists of a rectangular array of processing elements (PE) that are connected by a reconfigurable interconnect network. Each PE consists of a functional unit (FU) for word-wide arithmetic and logic operations, a register set, and a context memory that defines several configurations for the PE. A context is selected by a control unit which can vary significantly for the various architectures. So does the interconnect network and therefore both are not further specified in the general CRC model.

Based on that general specification we have mapped the operations and control structures of several applications described in C onto the CRC model. The model was successively augmented with features that enable the execution as proposed by the application mapping. These features are specified by a further refined model that is referenced as CRC-A in the following.

Similar to our approach, NEC architected the DRP “based on a clear picture of how C code is compiled into hardware” [3].

Fig. 1 depicts the features that are common to the DRP architecture and the CRC-A model. The interconnect network is subdivided into word-wide data channels and 1-bit status signals but not further detailed. For the FU and the registers, the word-wide data flow is also separated from the 1-bit signals. The output of the FU can be stored in a register and it can be fed into the interconnect network to execute two or more operations in a chain of FUs. For the control unit, it is only specified that it implements a finite state machine (FSM) controlled by the 1-bit status signals and that an entry of the context memory is selected by the FSM at the beginning of each clock cycle.

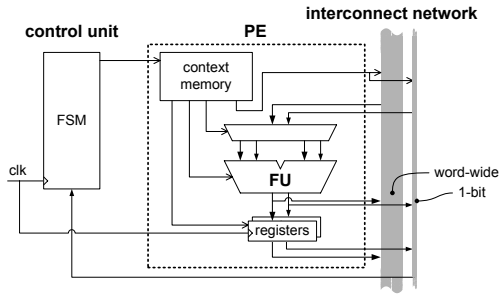


Fig. 1. Features common to the CRC-A model and the DRP architecture.

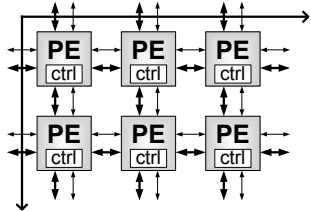


Fig. 2. Topology of the interconnect network and the control unit of the CRC-A model.

This refined CRC model specifies fewer features than the CRC-A model and does not cover all features of the DRP. Specifics for both architectures are discussed next.

#### A. CRC-A model

Fig. 2 shows the topology of the interconnect network and the control unit for the CRC-A model. Each PE features its own control unit and the interconnect network implements a nearest-neighbor (NN) network. The PEs at the borders of the array use the NN-network for I/O.

For the CRC-A model we have implemented a number of RT-level components (FUs with and without multiplier, NN-networks with one or two data channels, etc.) that can be combined in various ways so that different architecture instances can be synthesized and analyzed in a highly automated flow with reasonable effort. To create architecture instances, the CRC-A model is configured by selecting RT-level components and setting parameters of the Verilog code (e.g., the word-width and the number of contexts). This configuration at design time of the architecture is based on the requirements of execution schemes as described in Section III.

We do not claim that the features of the CRC-A model are most suitable for a wide range of applications. As a matter of fact, the CRC-A model has simply not exposed serious bottlenecks for the applications that we considered so far, and it is in particular suitable with regard to our automated synthesis and analysis flow.

#### B. NEC-DRP

For the DRP architecture, there is one control unit (“state transition controller”) for an array of 8x8 PEs as depicted in Fig. 3. The array is surrounded by embedded memory blocks. The interconnect network implements a more sophisticated

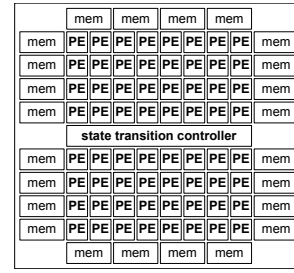


Fig. 3. A tile of the DRP architecture [3].

topology compared to the simple NN-network of the CRC-A model. In reference to the CRC model depicted in Fig. 1, the word-width of the DRP is 8 bits and there are 16 contexts. A DRP “core” is composed of one or more such DRP “tiles”. For our validation of execution schemes, we used the DRP-1 prototype core that consists of 8 tiles, i.e. 512 PEs in total.

### III. EXECUTION SCHEMES

For the design of execution schemes, it is assumed that the input data of the application arrives at a certain rate (data input rate, DIR) appointed by the surrounding environment and/or memory accesses. This performance constrained approach reflects a system designer’s point of view who typically fixes the needed bandwidth or the available memory of a system or varies it only within tight limits.

For a variety of DIRs, the operations of the application are scheduled for execution so that the required throughput is achieved using a minimal number of FUs. Minimizing the number of FUs can be accomplished by assigning the operations to different contexts if the DIR allows distributing the operations over multiple clock cycles. In addition to that, branches in the control flow can be handled by context switches rather than incorporating them in the data path. Such a multi-context execution does not only allow it to re-use the FUs but also to re-use the interconnect network in different clock cycles.

After scheduling, the execution of the application is pipelined if necessary, registers are allocated, and an FSM that controls the execution is extracted. The result of all these steps defines an execution scheme and hence how many FUs, registers, I/O ports, states, and contexts are needed to execute an application under a given DIR constraint. The execution scheme also defines the features required for the control unit.

Further details on execution schemes are presented in [5]. In the remainder of this section, techniques that we use for generating execution schemes are discussed. In the next section these techniques will be validated by mapping two examples to the DRP-1 prototype.

#### A. Techniques for Data Flow

The most interesting aspect of processor-like reconfigurable architectures is certainly *multi-context execution* since it provides an additional degree of freedom for application mapping compared to statically reconfigurable architectures. Early work

on fast reconfiguration has been presented by DeHon [6] and Trimberger e.a. [7]. But the principle of multi-context execution is actually well-known for much longer from von Neumann architectures where the context is changed with each instruction. To generate a simple multi-context execution scheme, we schedule all instructions that can be executed in parallel for execution in one control step. Each of these control steps is assigned to a separate context and the contexts are executed sequentially.

But the instruction level parallelism of C descriptions is usually too low to satisfy the DIR constraint of streaming applications and loop kernels. We use *pipelining* to increase the parallelism for such applications as it is commonly done for statically reconfigurable architectures.

If the DIR allows distributing the operations over multiple clock cycles, we combine the previous techniques as *multi-context pipelining*. This corresponds to the software pipelining techniques used by software compilers. Mei e.a. present a modulo scheduling algorithm for mapping loop kernels to coarse-grained reconfigurable architectures in [8].

To reduce the latency of processing an input sample as well as the register requirements, *chaining* of operations is a well-known technique from hardware design that we combine with the above techniques. While chaining is fully supported by fine-grained FPGAs, it can not be realized by software processors and often not by coarse-grained reconfigurable architectures.

### B. Techniques for Control Flow

Branches in the control flow can be resolved by *spatially multiplexing* the data path. This transforms the control flow graph of an application into a pure data flow graph that can be further processed as described above. This approach is commonly used for reconfigurable architectures, e.g. by Huang and Malik [9], and is also well-known from hardware design.

Alternatively, the branches can be assigned to different contexts, i.e. temporally multiplexed, to minimize the number of required FUs without impact on the performance since always only one of the branches is actually needed during execution. We call this technique *multi-context control flow branches* which corresponds to the conditional jump instructions of von Neumann architectures.

As an extension of the previous technique we use *pipelined multi-context control flow branches*. Using this technique, a pipeline stage may change its state and context depending on the results of a previous stage. If the target architecture features only one control unit for all PEs, an excessive number of states can be required to ensure a sustained DIR for all combinations of branches in the application. The CRC-A model, featuring a control unit in each PE, can implement independent FSMs for each pipeline stage so that the number of states and contexts is minimal since the possible combinations haven't to be considered at compile time.

## IV. EXPERIMENTS

To validate the techniques for data flow, we used the fifth order elliptical wave filter from the high-level synthesis

benchmark suite [10]. It consists of 26 16-bit add operations without branches in the control flow.

A multi-context execution scheme and a combination with chaining were obtained by simply setting the clock period constraint as appropriate. To realize pipelining, the DRP compiler provides an automatic scheduling mode with manual pipelining support. In this mode, the pipeline stage boundaries have to be specified manually by inserting C macros provided for this purpose. We used these macros to obtain a pipelined execution scheme and a combination of pipelining and chaining. To achieve multi-context pipelining, additionally the source code had to be rearranged to define which of the stages are executed concurrently. We implemented multi-context pipelined execution schemes that use 2, 4, and 8 contexts.

To validate the techniques for control flow, the loop kernel of a RGB to CMYK conversion is used. This example is part of the EEMBC Consumer Benchmark [11] and contains three if/else statements:

```
c=255-r; m=255-g; y=255-b;
if (c<m) { k=(c<y)?c:y; }
else     { k=(m<y)?m:y; }
c=c-k; m=m-k; y=y-k;
```

By using clock constraints and the pipelining macros, we realized a multi-context, a chained, and a pipelined execution scheme. For these schemes, the DRP compiler resolved the control flow by inserting three multiplexers. Although the DRP compiler generates multi-context control flow branches if necessary to accommodate larger applications, we had to rearrange the source code to enforce this execution scheme for a smaller example. In particular, we serialized the nested branches and moved the calculation of *c*, *m*, and *y* into the branches to remove redundant operations. For combination with pipelining, and thus reading new input values *r*, *g*, and *b* at every clock cycle, we had to modify the source code extensively. We distributed the pipeline stages over three independent processes communicating via the memory blocks depicted in Fig. 3 so that each process is controlled by its own state transition controller.

## V. RESULTS

Fig. 4 shows the relationship between performance and resource usage for the execution schemes. The dots (connected by solid lines) represent the execution schemes that were realized by just setting compiler constraints or inserting C macros in the source code. The squares (connected by dashed lines) represent the execution schemes that required modifications of the source code. Fig. 4 a) and c) show the number of required FUs for each execution scheme. Fig. 4 b) and d) show the number of required PEs which additionally takes into account the required registers.

For the execution schemes involving pipelining, the number of PEs is significantly higher than the number of FUs (different scales are used in the figures). This is due to the fact that a significant number of PEs is needed to pass intermediate

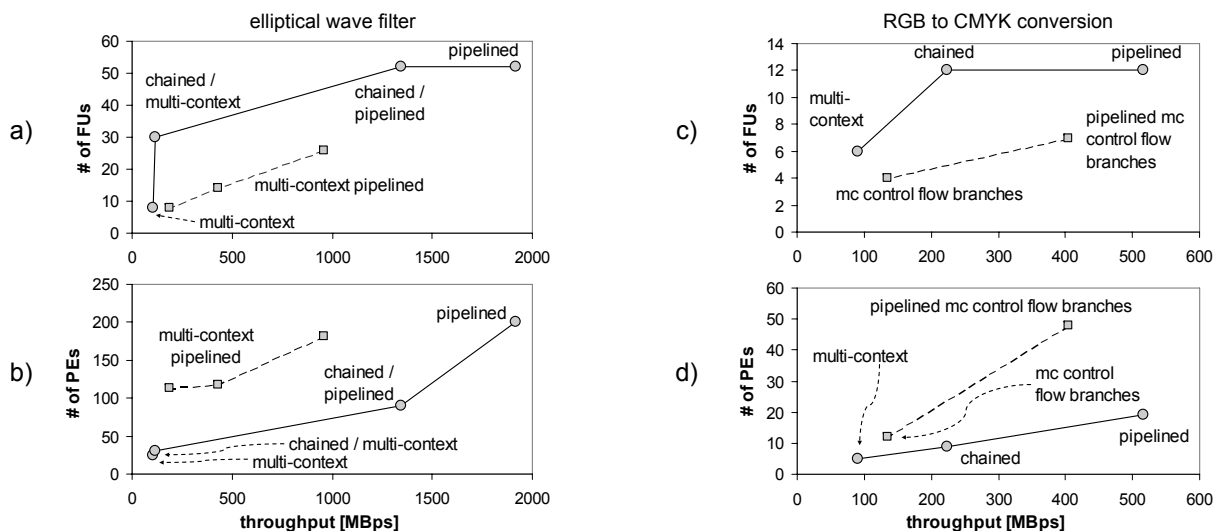


Fig. 4. FU and PE usage for the example applications.

results to the subsequent pipeline stage without being part of the calculation.

In our approach to designing application domain specific instances of the CRC model, we develop instances so that the number of required PEs is reasonably close to the number of required FUs as in Fig. 4 a) and c). E.g., when targeting applications that require high throughput we include abundant registers in the PEs to achieve a good performance-to-cost-ratio. Correspondingly, we focus on multi-context pipelining and (pipelined) multi-context control flow branches rather than chaining, since these techniques provide the best results in between the two extremes given by multi-context and pipelined execution.

In contrast to the CRC model, the DRP-1 prototype is a fixed architecture that is supposed to be suitable for a wide range of applications and the compiler must find a good solution for this particular architecture. As Fig. 4 b) and d) show, the execution schemes that required code modifications yield suboptimal results. In particular, pipelined multi-context control flow branches required excessive resources since the application must be distributed over multiple DRP tiles to realize one control unit for each pipeline stage. This execution scheme also required the most source code transformations.

## VI. CONCLUSIONS AND FURTHER WORK

The experiments show that the execution schemes proposed for the CRC model can be realized by a commercial architecture and its associated C compiler. A general conclusion on which architecture or execution scheme is best can not be drawn. The results rather demonstrate the importance of providing a compiler that is well coordinated with the architecture. Based on our experiments, we believe that this was done well for the DRP architecture. Independent of the architecture model, the results show how processor-like reconfiguration can be deployed to flexibly trade-off performance and resource requirements.

In ongoing research, we use our design environment to develop architectures that are specialized on application domains or provide advanced features like temporal-spatial voltage scaling [12]. The research also includes comparisons to other reconfigurable architectures including FPGAs.

## ACKNOWLEDGMENT

This work is funded by DFG under RO-1030/13 within the ‘Priority Program 1148’ which is focused on reconfigurable computing systems

## REFERENCES

- [1] T. Oppold, T. Schweizer, T. Kuhn, W. Rosenstiel, U. Kanus, and W. Straßer, “Evaluation of ray casting on processor-like reconfigurable architectures,” in *Proc. FPL’05*, 2005.
- [2] T. Oppold and W. Rosenstiel, “Evaluation and design of processor-like reconfigurable architectures,” in *Proc. FPL’06*, 2006.
- [3] M. Motomura, “A dynamically reconfigurable processor architecture,” in *Microprocessor Forum*, 2002.
- [4] K. Wakabayashi and T. Okamoto, “C-based SoC design flow and EDA tools: An ASIC and system vendor perspective,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1507–1522, 2000.
- [5] T. Oppold, T. Schweizer, J. Oliveira Filho, S. Eisenhardt, T. Kuhn, and W. Rosenstiel, “Execution schemes for dynamically reconfigurable architectures,” in *Proc. SASIMI’06*, Nagoya, Japan, 2006.
- [6] A. DeHon, “DPGA utilization and application,” in *Proc. FPGA’96*, 1996.
- [7] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, “A time-multiplexed FPGA,” in *Proc. FCCM’97*, 1997.
- [8] B. Mei, S. Vernalde, D. Verkest, H. DeMan, and R. Lauwereins, “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling,” in *Proc. DATE’03*, 2003.
- [9] Z. Huang and S. Malik, “Exploiting operation level parallelism through dynamically reconfigurable datapaths,” in *Proc. DAC’02*, 2002.
- [10] Benchmarks for the 1992 High Level Synthesis Workshop, <http://ftp.ics.uci.edu/pub/hlsynth/HLSynth92>.
- [11] Embedded Microprocessor Benchmark Consortium (EEMBC), <http://www.eembc.org>.
- [12] T. Schweizer, J. Oliveira Filho, T. Oppold, T. Kuhn, and W. Rosenstiel, “Evaluation of temporal-spatial voltage scaling for processor-like reconfigurable architectures,” in *Euro DesignCon*, 2005.