Efficient Mapping and Functional Verification of Parallel Algorithms on a Multi-Context Reconfigurable Architecture

Markus Rullmann, Dresden University of Technology, Dresden, Germany
Sebastian Siegel, Dresden University of Technology, Dresden, Germany
Renate Merker, Dresden University of Technology, Dresden, Germany
Julio A. Oliveira Filho, University of Tübingen, Tübingen, Germany
Thomas Schweizer, University of Tübingen, Tübingen, Germany
Tobias Oppold, University of Tübingen, Tübingen, Germany
Wolfgang Rosenstiel, University of Tübingen, Tübingen, Germany

Abstract

Parallel multi-context reconfigurable architectures provide very attractive platforms with respect to computational performance and reconfigurable features. Today's challenge is the exploitation of this reconfigurable and computational potential to ascertain efficient solution for mapping applications onto these architectures. The demand for appropriate tools is evident.
In this paper we provide a combination and a mutual adaption of two separate tools to create a continuous design flow for parallel multi-context reconfigurable architectures. Especially we present the interaction of a parameterized mapping tool for mapping compute intensive algorithms on processor arrays and a subsequent verification of the mapping results using the Configurable Reconfigurable Core (CRC) architecture model. The SystemC implementation of the CRC model leads to a cycle accurate functional simulation of the realization.
Using this continuous design flow we derive an efficient realization of the edge detection algorithm (EDA) on a parallel multi-context reconfigurable architecture. We describe in detail how the parallel realization of the EDA has to be translated in a specification for programming the CRC model.

1 Introduction

Parallel multi-context reconfigurable architectures are adequate target platforms for implementations of compute intensive applications. Nowadays there is a high demand for tools assisting the mapping of the application under resource constraints onto the architecture, proving the correctness of the mapping and evaluating the mapping. In this paper we present the interaction of both the mapping of compute intensive algorithms on parallel multi-context reconfigurable architectures and the verification of the mapping results by simulation. For mapping we developed a parameterized partitioning and mapping strategy leading to efficient realizations of algorithms on parallel processor arrays [11]. In particular, an optimization tool receives the parameters of the architecture, such as the interconnect channels and functional units, and generates the efficient realizations [12, 13]. The mapping techniques complements the scheduling methods in [9], in which parallelism is exploited with pipelining the allocation of multiple PE to a single computational kernel only. In this work, we apply our recently introduced mapping methods [12, 13] to a reconfigurable architecture. After mapping, a functional verification and evaluation of the realizations has to be performed. We developed a Configurable Reconfigurable Core (CRC) architecture model as a general and parameterizable model for coarse-grained reconfigurable arrays [7]. This CRC model provides us with an ideal environment for the functional verification of the derived realizations due to the following reasons:

- The architectural parameters are configurable.
- It provides a SystemC model at a cycle accurate, but high abstraction level.
- It provides a Verilog model to derive area, speed and energy consumption.
- The CRC architecture resembles our formally modeled processor array very closely.

Furthermore, the simulation at system level with the CRC model leads to information about the behavior of the realizations in complex environments.
In this paper we apply our approach to the edge detection algorithm (EDA). We describe the derivation of an efficient realization of the EDA with our partitioning and mapping strategy under consideration of the constraints of the reconfigurable target architecture. Especially we show how this mapping can be scaled to processor arrays of different size and how to maintain a high utilization of the functional units. The derived realization of the EDA is used as specification for programming the CRC model. The SystemC implementation of the CRC model leads to a cycle accurate, yet fast, functional simulation of the realization. Further simulations of the EDA realization in Verilog RTL or gate level can evaluate performance, power and area of the implementation.

The paper is organized as outlined below. We briefly describe our mapping approach in Section 3. At first we give a summary of the notation of an algorithm as uniform recurrence equation. We continue to describe a partitioning strategy to match the processor array parameters of the intended target architecture. We further describe the model employed to map the algorithm to the processor array in full detail. In Section 4, we give a brief overview of the CRC architecture and of the operation of the reconfigurable processing elements (PE). In this section we also discuss extensions of the original CRC PE that allow more efficient mappings of parallel programs. Finally, in Section 5, we present an efficient mapping of an Edge Detection Algorithm on an instance of the CRC model. The mapping is scalable to allow implementations with different throughput/area trade-offs.

2 Related Work

Only a few compilation tools and application-mapping techniques have been presented that exploit the abundant computation resources found in coarse-grained reconfigurable architectures. Even a fewer number of them deal with the realization of parallel algorithms. GUI-based design tools, such as Morphosys' *mView* [15] obviously limit the size of the design to be handled. The compiler DRESC [5], developed for the architecture ADRES, exploits loop level parallelism by applying a variation of modulo scheduling. It considers the modulo scheduling as a placement in a 3D resource array. Like in our approach, such formulation allows the consideration of routing resources during the mapping. However, they have some limitations to handle some architecture constraints, such as pipelined FUs and limited register files.
An exact optimal solution for the scheduling, binding and routing problem in presented in [1] for mapping applications in an array of processing elements. Although an optimal solution is always found if it exists, the problem was formulated considering an specific interconnection network (nearest-neighbors). It may then occur, that an application suffer penalties during mapping due to the insufficient routing resources. Similar to this work, they also use the CRC Model as a basis for mapping of applications and verification of results. Additionally, some commercial tools are available, such as the DRP from NEC [6] and HiveCC [14] from SiliconHive(Philips). The DRP was developed based on the hardware synthesis tool Cyber. It uses BDL, a C-based behavioural description language, that allows the explicit description of parallelism between instructions. Alternatively, the HiveCC compiler accepts ANSI C and automatically extracts parallelism of the code. None of the above cited approaches focuses on parallel algorithms. A methodology for mapping parallel algorithms specified in a polytope model to coarse grained arrays is introduced e. g., in [2]. However, the authors remain unclear how to efficiently exploit the interconnect architecture with a general method. Our tool optimizes simultaneously routing, register and ALU usage. In this paper, we demonstrate how the tool can be used to implement parallel algorithms on the CRC architecture.

3 Algorithm Partitioning and Mapping

3.1 Algorithm Notation and Partitioning

In this paper we consider algorithms that may be described as a system of uniform recurrence equations (SURE). This notation implies a parallel execution model in contrast to sequential nested loop programs. A SURE consists of a set of J statements ($1 \leq j \leq J$) of the form:

$$S_j : \; y_j\,[\mathbf{i}] := f_j\big(\ldots, y_i\big[\mathbf{i} - \mathbf{d}^r_{j,i}\big], \ldots\big), \forall \mathbf{i} \in \mathcal{I}_j,\; i,j \in \mathbb{N}\,,$$

where $\mathbf{i}$ denotes an *iteration* in the iteration space $\mathcal{I}_j$ of statement S_j, $\mathcal{I}_j$ is a polyhedral subset of a $\mathbb{Z}$-module and f_j denotes a single-valued function.
A variable y_i that is computed by statement S_i is a *dependent variable* if it is input to some (other) statement S_j. Vector $\mathbf{d}^r_{j,i}$ denotes the corresponding uniform data dependency. Upper index $r \in \mathbb{N}$ is used only if more than one data dependency exists between statements S_j and S_i. A variable y_i that appears only on the right side of one or more statements denotes an *input variable* to the algorithm.
With the embedding given by $\mathcal{I} = \mathrm{conv}(\bigcup_j \mathcal{I}_j) \subset \mathbb{Z}^n$ we determine the iteration space $\mathcal{I}$ of the SURE. In Algorithm 1 we show the edge detection algorithm (EDA) in SURE notation. The EDA is very similar to the Sobel filter used in many image processing applications. The parameters N and M denote the width and height of the input image, which resides in variable $p_i\,[\,{}^{x}_{y}\,]$. Variable $s\,[\,{}^{x}_{y}\,]$ will receive the filtered output image.
The SURE notation implies a processor array (PA) with the size of $\mathcal{I}$, e. g., for every iteration $\mathbf{i} \in \mathcal{I}$ there exists one processing element (PE) that executes all S_j. In order to compute the EDA on a target architecture with a limited number of PEs, we employ LPGS partitioning [11]. In this method, the iteration space is partitioned into rectangular tiles of equal size. All iterations within a tile are executed in parallel, one iteration $\mathbf{i}$ per PE. LPGS-partitioning separates an iteration $\mathbf{i} \in \mathcal{I}$ into $\widehat{\boldsymbol{\kappa}}$ (denoting a partition) and $\boldsymbol{\kappa}$ (representing the position within a partition) by the tiling step as follows:

$$\mathbf{i} = \boldsymbol{\Theta}\widehat{\boldsymbol{\kappa}} + \boldsymbol{\kappa}, \qquad \begin{cases} 0 \leq \kappa_k < \vartheta_k,\; 1 \leq k \leq n, \\ \widehat{\boldsymbol{\kappa}} \in \widehat{\mathcal{K}} \subset \mathbb{Z}^n,\; \boldsymbol{\kappa} \in \mathcal{K} \subset \mathbb{N}^n \end{cases},$$

where $\boldsymbol{\Theta} = \mathrm{diag}(\vartheta_1 \cdots \vartheta_n) \in \mathbb{N}^{n \times n}$ is a square matrix whose diagonal elements represent the size of the

Algorithm 1: Edge detection algorithm (EDA)

$$
\begin{array}{lrl}
S_1: & p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = p_i\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_1 \\
S_2: & q\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = 2 \cdot p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_2 = \mathcal{I}_1 \\
S_3: & h_1\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = p\left[\begin{smallmatrix}x\\y-2\end{smallmatrix}\right] + p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_3 \\
S_4: & h_2\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = h_1\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] + q\left[\begin{smallmatrix}x\\y-1\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_4 = \mathcal{I}_3 \\
S_5: & v_1\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = p\left[\begin{smallmatrix}x-2\\y\end{smallmatrix}\right] + p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_5 \\
S_6: & v_2\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = v_1\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] + q\left[\begin{smallmatrix}x-1\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_6 = \mathcal{I}_5 \\
S_7: & h_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = h_2\left[\begin{smallmatrix}x-2\\y\end{smallmatrix}\right] - h_2\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_7 \\
S_8: & h_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = |h_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]|, & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_8 = \mathcal{I}_7 \\
S_9: & v_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = v_2\left[\begin{smallmatrix}x\\y-2\end{smallmatrix}\right] - v_2\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_9 = \mathcal{I}_7 \\
S_{10}: & v_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = |v_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]|, & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_{10} = \mathcal{I}_7 \\
S_{11}: & s\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = h_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] + v_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], & \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_{11} = \mathcal{I}_7
\end{array}
$$

with

$$
\mathcal{I}_1 = \{\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{Z}^2 \mid \begin{smallmatrix}0\le x<N\\0\le y<M\end{smallmatrix}\},\quad \mathcal{I}_3 = \{\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{Z}^2 \mid \begin{smallmatrix}0\le x<N\\2\le y<M\end{smallmatrix}\},
$$
$$
\mathcal{I}_5 = \{\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{Z}^2 \mid \begin{smallmatrix}2\le x<N\\0\le y<M\end{smallmatrix}\},\quad \mathcal{I}_7 = \{\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{Z}^2 \mid \begin{smallmatrix}2\le x<N\\2\le y<M\end{smallmatrix}\}
$$

partitions in each of the n directions of the iteration space $\mathcal{I}$.

We will illustrate the LPGS partitioning for an architecture with 5×4 PE, e. g., $\mathbf{\Theta}^{54} = \left(\begin{smallmatrix}5&0\\0&4\end{smallmatrix}\right)$. All iterations inside a tile are executed in parallel on the PA. The tiles are processed sequentially on the PA, until all iterations of the algorithm are finished.

Partitioning also has an effect on the data dependencies. In our model, uniform data dependencies are realized by uniform routing. However, due to the tiling of the iteration space, some data needs to be stored between partitions. We discriminate between intra- and inter-tile data dependencies. Consider S_7 at iteration $\mathbf{i} = \left[\begin{smallmatrix}4\\1\end{smallmatrix}\right]$, hence the computation of $h_3\left[\begin{smallmatrix}4\\1\end{smallmatrix}\right]$. The statement has an intra-tile dependency (Fig. 1(a), gray arrow), which is routed inside the PA between the corresponding PEs (Fig. 1(b), gray arrow). Now consider $h_3\left[\begin{smallmatrix}5\\1\end{smallmatrix}\right]$, computed in tile $\widehat{\boldsymbol{\kappa}} = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ at PE $\boldsymbol{\kappa} = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$: the source of dependency $\mathbf{d}^1_{7,4} = \left(\begin{smallmatrix}2\\0\end{smallmatrix}\right)$ (Fig. 1(a), black arrow) is in tile $\widehat{\boldsymbol{\kappa}} = \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ of PE $\boldsymbol{\kappa} = \left(\begin{smallmatrix}3\\1\end{smallmatrix}\right)$. This inter-tile data dependency does not remain inside one partition and requires a routing from PE $\boldsymbol{\kappa} = \left(\begin{smallmatrix}3\\1\end{smallmatrix}\right)$ to PE $\boldsymbol{\kappa} = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ (Fig. 1(b), black arrow). However, the data can be routed the same way as the intra-tile dependency with one extension: the data leaves the PA on the right side (for partition $\widehat{\boldsymbol{\kappa}} = \left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$) and is written back to the left side of the PA when partition $\widehat{\boldsymbol{\kappa}} = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ is executed, cf. Fig. 1(b) (dotted arrow). This method is beneficial in many ways:

- uniform modeling of all dependencies regardless of their position in the PA,
- uniform control for all PEs,
- small PE internal memory requirements, since the data is store outside the PA for later use.

The memory requirements to store the inter-tile data dependencies can be controlled by an appropriate partitioning.

3.2 Architecture Model

In order to derive a detailed mapping of the algorithm, we need to model the key elements of a possible target architecture. The model is very generic and can be configured for different target architectures. In particular the following elements are modeled:

Functional Units

Functional units (FUs) perform the operations given by the SUREs statements. Each processing element (PE) can contain instances of FU types, each FU type can perform a defined set of operations. Currently, we model one FU per PE that has a fixed latency of 1 cycle. We are working on a more general model, in which each operation on a FU has a latency of n cycles. New operations can be started m cycles after the previous operation. If $m < n$ then the FU operates in a pipelined fashion.

Registers

Registers are used for storing data in PEs. The storage is necessary to provide FU results or data from channels to consecutive operations and data transfers. Registers are not modeled directly, instead the final schedule defines a variables' storage cycle and read cycles for dependent operations and data transfers. Nevertheless the total number of required registers is considered in the optimization.

Processing Elements

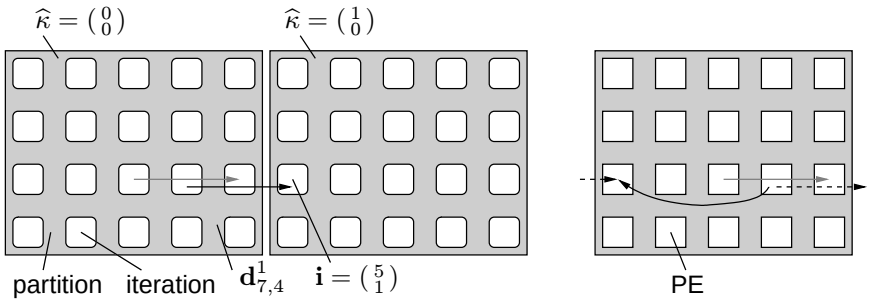
A processing element is an entity containing FUs and registers as described before. The intended target architecture for the processor array consists of a regular one- or two-dimensional array of equal PEs.

Channels

Communication channels connect the processing elements (PEs) with each other and with the periphery of the processor array. The channels are used to route data from the source PE to a target PE. We define a set $W = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{|W|}\}$ of channels $\mathbf{w}_i \in \mathbb{Z}^2$. To each element $\mathbf{w}_k$ of W corresponds a latency $l_k \in \mathbb{N}$ which represents the time required to transfer one data item on that channel. The mapping of the EDA requires only 2 channel $\mathbf{w}_1 = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ and $\mathbf{w}_2 = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$. The CRC architecture currently implements a nearest neighbor interconnect with $l_1, l_2 = 0$.

3.3 Communication and I/O Problem

The uniform data dependencies need to be realized by a conflict free organization of the data transfer they cause. The communication problem consists in minimizing the implementation cost in terms of channels and registers for these data dependencies. In order to determine this cost, we introduce a model which allows a description of the communication [12].



(a) Iteration Space (b) Processor Array

Figure 1 Routing of intra- and inter-tile dependencies depicted by gray and black arrows, respectively. The dependencies are shown in the iterations space (a) and on the PA (b).

Similar to the communication problem we introduced the I/O problem [13]. I/O is caused by the non-uniform input and output statements of the algorithm. The I/O data and the data transfer caused by data dependencies share the channels between the PEs and the I/O ports to the periphery.
Obviously, the solution to the problems depend on the target architecture. If the number of channels or functional units is not sufficient, the statements are scheduled with a larger iteration intervals. The iteration interval λ defines the length (in cycles) of the iteration period. However larger iteration intervalls may increase the number of required registers in a PE. If the number of registers in the target architecture is not sufficient for a given algorithm, the mapping will fail. However, we think that this is a minor restriction in practice since the SUREs have usually few operations and thus require few registers.

Tool Support

We have implemented a tool that solves the communication and an I/O problem for a given algorithm. The tool receives the SURE parameters, e.g. the statements and the data dependencies. The tool builds a model that is solved using a commercial optimization tool [3]. Our tool does not support direct input of a high-level language since general SUREs cannot be expressed efficiently in sequential programming languanges.
We introduced a model that allows us to solve the communication and I/O problem either within a single model or as a two step solution: at first the communication problem and secondly the I/O problem is solved using the constraints from the first solution.
The model assigns a schedule for a pipelined operation of the PEs. The statements of one iteration may be distributed over one or more such periods.
A solution to the communication and I/O problem specifies the following for the data dependencies and statements of the SURE:

- a schedule for the data transfer via the given channels,
- a schedule for the operation of each statement,
- storage and access cycles for each variable.

Currently, the tool output must be translated into program code for the context memory of the CRC model manually.

4 CRC Architecture

A complete functional verification of our partitioning and mapping strategy was possible through simulation in the CRC architecture model. The CRC is a general and parameterizable model for coarse-grained reconfigurable arrays. In its higher abstraction level, it describes a two-dimensional array of coarse-grained processing elements (PEs), surrounded by an interconnection network, as depicted in Figure 2.

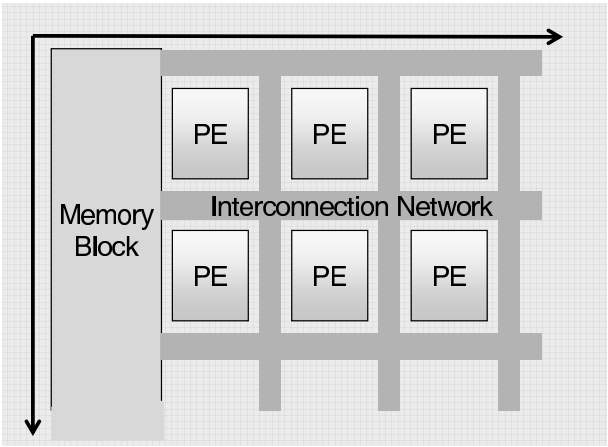


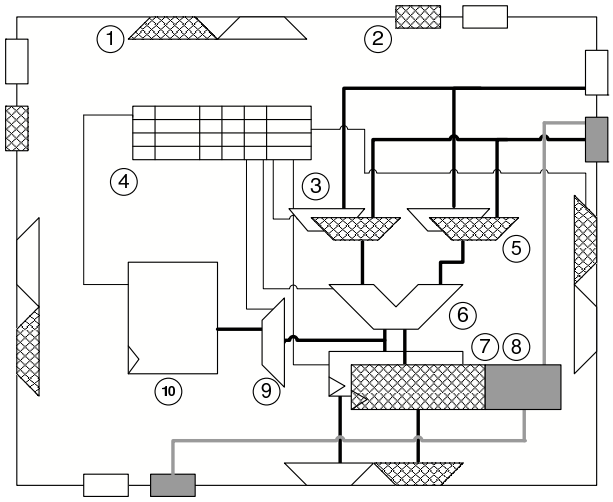
Figure 2 CRC Model - basic view.

As mentioned before, CRC stands for Configurable Reconfigurable Core. It is configurable because, during the architecture design time, it is possible to adjust parameters of the model in order to customize one architecture instance. Typical parameters are the array geometry (i.e. its length and width), type of the interconnection network, type of the PE functional

unit, number of registers and number of reconfiguration contexts. We easily configured one architecture instance according to our needs, adjusting specially the aspects discussed in section 3.2. The details for the design of such architecture instance are discussed in the next section.
Additionally to the configurability of the core, the CRC uses a *processor-like reconfigurable* scheme [8]. In opposition to FPGAs, whose reconfiguration phase takes several clock-cycles, *processor-like reconfiguration* applies reconfiguration at a cycle by cycle basis. That allows the instantiation and execution of exactly that part of the circuit that is needed in the current clock-cycle. Such fast reconfiguration scheme was shown to increase performance when the architecture is targeted to and application domain such as computer vision [9], and to introduce new power optimization possibilities [10].
The CRC Model was described in two different abstraction levels: a Verilog Model and a transaction-level cycle-accurate SystemC model. The first variant is intended for synthesis, and though it is adequate for area, timing and energy consumption evaluation. At this description level, simulation at RTL and gate level are also possible, but it may take long time even for small applications. The Verilog Model is the more adequate description for later stages of the design process where more realistic evaluations must be acquired.
Besides the Verilog model, a SystemC description is available, which provides a more flexible and adequate alternative for the early architectural exploration and functional verification. It allows, along with the normal parametrization, an easier inclusion of new elements, such as additional I/O ports and register banks in a PE. It can also be integrated and simulated in most System-On-Chip design platforms which accept SystemC. Such variant suited to our needs, as we intended to assert our methodology through a functional verification of the edge detection algorithm example. Therefore, we chose to use the SystemC description of CRC Model, which is discussed in details in the following section.

4.1 CRC SystemC Model

The CRC SystemC Model is a library of parameterizable components described in SystemC. Such components model common coarse-grained structures such as multiplexers, register banks, context memory, finite state machines and functional units. It is possible to adjust individually the number of input and output ports for each component, and some of them have a preset of associated behaviors. The finite state machine, for example, can be configured to describe a Mealy, a Moore or a Medvedev variant.
An standard interface for the functional unit allows the inclusion of new operations described in C/C++. Complex functionalities may be programmed in a software-like style. That allows to quickly evaluate the performance impact in adopting few complex FUs compared to composing many simpler FUs to realize one complex operation. The SystemC model also offer an standard mechanism for input of stimuli data and observation of generated values.
Transaction channels and elements were implemented using a combination of SystemC and C++ templates, in a similar way as proposed in [4]. Such template approach allows the data type customization and higher reusability of the modules.
As a start point for customizing the CRC Model, we used a pre-modeled PE which is delivered with the library. That pre-modeled PE has the same structure of the PE instance presented in [7]. We may then assume the evaluation of area, power and timing presented in this previous work as a first approximation for our real architecture. We explain the processing element modeled here, including the modifications we did in details.



1 Output Multiplexer
2 Input Port
3, 5 Operand Multiplexer
4 Context Memory
6 Functional Unit
7, 8 Register Set
9 Flag Multiplexer
10 Finite State Machine

Figure 3 Processing element modeled using the CRC-SystemC Model.

Each PE was modeled according to the figure 3. For sake of simplicity, not all connection channels were depicted and elements that work over data (flag) are designed in dashed (blank) style. Basically, the PE works in the following way:

1. At each positive transition of the clock signal, the finite state machine (10) and the register set (7,8) are activated. The finite state machine goes to its next state, based on the information presented by the flag multiplexer (9). This new state determines which context will be selected in the context memory (4). The register

set stores, simultaneously the result of the last operation.

2. The information for the selected context is passed to the context memory (4), which outputs accordingly all the control signals to each element in the PE datapath. Though, choosing a new context reconfigures the datapath to execute another functionality.
3. After the control information reaches each PE element, the data path is ready to execute. Operand multiplexers (3,5) are commanded to select the data/flag line that should be operated by the FU (6). These lines come from the register set (one per register in the set) and from the input ports (one per input port). The ALU receives the command that determines which operation is to be executed. The data/flag register set (7,8), is controlled according to the place where the result should be stored (see step 1). The flag line (ALU output and input ports) are also sent to the flag multiplexer (9). Based on that information, the next state of the state machine will be chosen.
4. The output multiplexers (1) are also controlled through the context memory. Each output multiplexer have as input the signals coming from the remaining three (input) ports and the signals coming from the register set (one per register in the set). The signals from other ports are selected when routing the information from one PE to another. Note that, while performing this routing, it is still possible to execute an operation in the ALU.
5. When the computation within one PE is finished, the system waits for the next clock cycle and restarts the process.

We customized for the architecture such that the input ports in the south and west side of a PE were directly connected to an extension of the register bank. In figure 3, such extension is depicted in gray. That modification allows data coming through these two input ports to be directly recorded in the registers without having to be processed by the FU. Another side effect of such modification is that an operation may be concurrently executed at the FU, while data is stored on the register bank.

5 Example: Edge Detection Algorithm

The EDA is shown in Algorithm 1. The algorithm can be implemented on a simple instance of the CRC architecture, because only ALU operations are used. Nevertheless a few modifications of the algorithm are necessary before the statements match the CRC's ALU operations: Because the CRC instance used does not implement a multiplication, statement S_2 is rewritten as:

$$S_2: \qquad q\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] + p\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], \qquad \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_2.$$

The absolute value operation in statements S_8 and S_{10} are realized using the overflow flag result from the operations S_7 and S_9, respectively. The CRC architecture state machine allows the conditional branching required to execute the rewritten statements given below:

$$S_8: \quad h_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = \begin{cases} -h_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] & \text{if } h_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] < 0 \\ h_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] & \text{else} \end{cases}, \quad \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_8 = \mathcal{I}_7$$

$$S_{10}: \quad v_4\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] = \begin{cases} -v_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] & \text{if } v_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] < 0 \\ v_3\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right] & \text{else} \end{cases}, \quad \left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathcal{I}_{10} = \mathcal{I}_7$$

Note that the algorithm is not a SURE anymore, but it can be treated equally to the sure since the data dependencies are still uniform and the data dependend operations are executed exclusively on the same FU.

5.1 Mapping Variants for a Scalable Processor Array

In this section we will present several realizations of the EDA. The variants differ in partitioning and in the used CRC architecture instance. At first the effect of partitioning and algorithm size on memory requirements is discussed.

We assume that the algorithm is uniform in all partitions, hence border effects are handled by an external controller rather than by the PEs. In our case this can be handled straightforward: the input image is extended such that perfect tiling is achieved. The output image is then contained in a subset of the output data.

The average communication- and I/O-rates depend on the selected partitioning. In addition the average communication rate also depends on the solution of the communication problem. The communication problem reduces redundancy in the data transfer between PEs if necessary, hence the communication is reduced as necessary. Due to the uniform nature of the communication inside the PA and on the border PEs, this also effects the communication rate at the periphery of the PA.

In Table 1, we present the average communication- and I/O rates of the solution for the partitioning with $\mathbf{\Theta}^{54}$. The solution is optimal for all horizontal communication and I/O and the vertical communication does not limit the size of the PA. The equations in Table 1 show an important property of the chosen partitioning: the uniform dependencies cause only I/O at the borders of the array. This I/O scales linear with the size of the PA in the same way as the total I/O

bandwidth of the PA increases with its size. The I/O rate of the EDA increases quadratic (by $\vartheta_1 \times \vartheta_2$) with the array size, but the I/O capabilities increase only linear. For the EDA, one can overcome this limitation in three ways: adding extra horizontal channels as ϑ_1 is increased, or increase only ϑ_2 and leave $\vartheta_1 = 5$ (which ensure max. use of the horizontal channel), or increase the iteration interval λ (which reduces ALU utilization).

	$\boldsymbol{\Theta}^{54} = \left(\begin{smallmatrix}5 & 0\\ 0 & 4\end{smallmatrix}\right), \lambda = 10$		$\boldsymbol{\Theta} = \left(\begin{smallmatrix}\vartheta_1 & 0\\ 0 & \vartheta_2\end{smallmatrix}\right), \lambda$	
	Req.	Max.	Req.	Max.
horizontal, data dep.	5×4	10×4	$5 \times \vartheta_2$	$\lambda \times \vartheta_2$
horizontal, I/O	5×4		$\vartheta_1 \times \vartheta_2$	
vertical, data dep.	5×5	10×5	$5 \times \vartheta_1$	$\lambda \times \vartheta_1$

Table 1 Parameters for the realizations depending on the partitioning size. The numbers given are the number of data transfer per iteration. The columns "Req." list the bandwidth required and "Max." the maximum bandwidth available for this PA size.

The average throughput is given in image pixels by the following equation:

$$\frac{\text{Pixel}}{\text{Cycle}} = \frac{\vartheta_1 \times \vartheta_2 \,\text{Pixel}}{\lambda \,\text{Cycle}} \tag{1}$$

In Table 2 we show the results for some possible realizations that are derived with the equations from Table 1. The efficiency of the realization is defined by the actual iteration interval λ divided by the smallest possible iteration interval $\lambda_{\min} = 10$. The peak external bandwidth defines the max. number of input data in one cycle.
The table shows the parameters of the solution solved originally by the optimization tool (case 1) and the parameters for two PAs with more PEs in the ϑ_1-dimension (cases 2 & 3). It can be seen, since the max. allowed bandwidth remains constant, the iteration interval must be increased to provide extra cycles for I/O. As a result, the efficiency of the PA decreases even though the algorithm throughput shows some improvement. In order to overcome the I/O limitations of the PA two solutions were possible: at first, the PA has been increased in height, which adds more horizontal I/O resources (case 4); secondly the PA has been increased in width and at the same time, a second horizontal channel is added to the PEs (case 5). Both solutions maintain an efficiency of 100% at the cost of higher allowed bandwidth. Solution (case 5) shows the same performance as (case 4) with a small additional cost for the extra horizontal channel.
The different realizations can now be used to simulate the different design points at system level with the CRC SystemC model. It provides us with valuable information about the behavior in complex environments, e. g., real-time performance. Later on, the synthesizable CRC model can provide accurate cost values in terms of energy, area and speed to the designer.

5.2 Detailed Mapping

In Table 3 the detailed specification of the realization for the partitioning with $\boldsymbol{\Theta}^{54}$ is shown. It has been derived directly from the result of our optimization problem. The table shows the scheduling for the operations with their operands, the register allocation, and the channel usage. The table contains the information for the PEs in the leftmost column of the PA as an example. Vector $\mathbf{i}$ represents the current iteration. Since we are using pipelining of the operations, the schedule contains also operations from the iteration $\mathbf{i}'$ that is executed on this PE before the current iteration and from the iteration $\mathbf{i}''$ that is executed on this PE after $\mathbf{i}$. The predecessor and successor can be determined by the globally sequential schedule and the tile associated with iteration $\mathbf{i}$.
The specification can now be translated into a program for the context memory and the state machine for the CRC model. In Figure 4, the state machine diagram is shown. The nodes are marked with the context selected in this state. For completeness, we also annotated the corresponding EDA statement. The machine starts in state C_0 as an initial state that is not part of the EDA. From that state, an infinite loop of instructions is entered. One iteration of the EDA is computed in every sequence running from C_1 to C_{12}. Note the coding of the data dependent execution of the rewritten statements S_8 and S_{10}: e. g., in C_7 the overflow flag of the ALU "$-$"operation is used to select the correct operation for S_8.
The program for the context memory is a translation of the specification given Table 3 into an equivalent control for the PE. The coding of the context memory is shown in Table 4. The context memory coding is explained on statement S_3 as an example: S_3 is stored in context C_2. Operand 1 is received by the PE from the vertical channel. The equivalent control is realized by setting DinMux1 to input South (S). In a similar manner all other control code is derived. The control signals are explained in Table 5.

Routing of Data Dependencies

The routing of data dependencies is given by the specification (Table 3). Since the routing is uniform, the routing schedule is the same for all PEs regardless of their position in the PA. We will illustrate the realized routing of the intra-tile dependency $\mathbf{d}^1_{7,4} = \left(\begin{smallmatrix}2\\0\end{smallmatrix}\right)$ for statement $h_3\left[\begin{smallmatrix}4\\1\end{smallmatrix}\right]$ (cf. Figure 1, gray arrow) already discussed in Section 3. The operation to compute $h_3\left(\begin{smallmatrix}4\\1\end{smallmatrix}\right)$ requires variable $h_2\left(\begin{smallmatrix}2\\1\end{smallmatrix}\right)$ computed at PE $\boldsymbol{\kappa} = \left(\begin{smallmatrix}2\\1\end{smallmatrix}\right)$. The scheduling defines that $h_2\left(\begin{smallmatrix}2\\1\end{smallmatrix}\right)$ is computed in C_5 and stored in register 1. In the next cycle

Case	PEs	ϑ_1	ϑ_2	Channels horizontal	Channels vertical	λ	Throughput Pixel/Cycle, (1)	Efficiency	Peak External Bandwidth
1	20	5	4	1	1	10	2	100%	4
2	40	10	4	1	1	15	2.66	66%	4
3	60	15	4	1	1	20	3	50%	4
4	40	5	8	1	1	10	4	100%	8
5	40	10	4	2	1	10	4	100%	8

Table 2 Throughput and efficiency computed for different PA realizations.

S_j	Operation	Reg 0	Reg 1	Reg 2	Reg 3	hor. Ch. out	hor. Ch. in	vert. Ch. out	vert. Ch. in
S_5	$v_1(\mathbf{i}) = p_i(\mathbf{i}+\binom{-2}{0}) + p_i(\mathbf{i})$	$p_i(\mathbf{i})$	$p_i(\mathbf{i}+\binom{-2}{0})$		$s(\mathbf{i}')$	$s(\mathbf{i}')$	$p_i(\mathbf{i}''+\binom{4}{0})$	$p_i(\mathbf{i})$	$p_i(\mathbf{i}+\binom{0}{-1})$
S_3	$h_1(\mathbf{i}) = p_i(\mathbf{i}+\binom{0}{-2}) + p_i(\mathbf{i})$	$p_i(\mathbf{i})$	$p_i(\mathbf{i}+\binom{0}{-1})$	$v_1(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{4}{0})$	$p_i(\mathbf{i}''+\binom{4}{0})$	$p_i(\mathbf{i}''+\binom{3}{0})$	$p_i(\mathbf{i}+\binom{0}{-1})$	$p_i(\mathbf{i}+\binom{0}{-2})$
S_2	$q(\mathbf{i}) = 2 \times p_i(\mathbf{i})$	$p_i(\mathbf{i})$	$h_1(\mathbf{i})$	$v_1(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{3}{0})$	$p_i(\mathbf{i}''+\binom{3}{0})$	$p_i(\mathbf{i}''+\binom{2}{0})$		
S_6	$v_2(\mathbf{i}) = q(\mathbf{i}+\binom{-1}{0}) + v_1(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{2}{0})$	$h_1(\mathbf{i})$	$v_1(\mathbf{i})$	$q(\mathbf{i})$	$q(\mathbf{i})$	$q(\mathbf{i}+\binom{-1}{0})$	$q(\mathbf{i})$	$q(\mathbf{i}+\binom{0}{-1})$
S_4	$h_2(\mathbf{i}) = q(\mathbf{i}+\binom{0}{-1}) + h_1(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{2}{0})$	$h_1(\mathbf{i})$	$v_2(\mathbf{i})$	$q(\mathbf{i}+\binom{0}{-1})$	$p_i(\mathbf{i}''+\binom{2}{0})$	$p_i(\mathbf{i}''+\binom{1}{0})$	$v_2(\mathbf{i})$	$v_2(\mathbf{i}+\binom{0}{-1})$
S_9	$v_3(\mathbf{i}) = v_2(\mathbf{i}+\binom{0}{-2}) - v_2(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{1}{0})$	$h_2(\mathbf{i})$	$v_2(\mathbf{i})$	$v_2(\mathbf{i}+\binom{0}{-1})$	$h_2(\mathbf{i})$	$h_2(\mathbf{i}+\binom{-1}{0})$	$v_2(\mathbf{i}+\binom{0}{-1})$	$v_2(\mathbf{i}+\binom{0}{-2})$
S_7	$h_3(\mathbf{i}) = h_2(\mathbf{i}+\binom{-2}{0}) - h_2(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{1}{0})$	$h_2(\mathbf{i})$	$v_3(\mathbf{i})$	$h_2(\mathbf{i}+\binom{-1}{0})$	$h_2(\mathbf{i}+\binom{-1}{0})$	$h_2(\mathbf{i}+\binom{-2}{0})$		
S_8	$h_4(\mathbf{i}) = \mathrm{abs}(h_3(\mathbf{i}))$	$p_i(\mathbf{i}''+\binom{1}{0})$		$v_3(\mathbf{i})$	$h_3(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{1}{0})$	$p_i(\mathbf{i}'')$		
S_{10}, S_1	$v_4(\mathbf{i}) = \mathrm{abs}(v_3(\mathbf{i}))$	$p_i(\mathbf{i}'')$		$v_3(\mathbf{i})$	$h_4(\mathbf{i})$	$p_i(\mathbf{i}'')$	$p_i(\mathbf{i}''+\binom{-1}{0})$		
S_{11}	$s(\mathbf{i}) = h_4(\mathbf{i}) + v_4(\mathbf{i})$	$p_i(\mathbf{i}'')$	$p_i(\mathbf{i}''+\binom{-1}{0})$	$v_4(\mathbf{i})$	$h_4(\mathbf{i})$	$p_i(\mathbf{i}''+\binom{-1}{0})$	$p_i(\mathbf{i}''+\binom{-2}{0})$		

Table 3 Specification for the realization of the EDA with $\mathbf{\Theta}^{54}$ for the leftmost PE column.

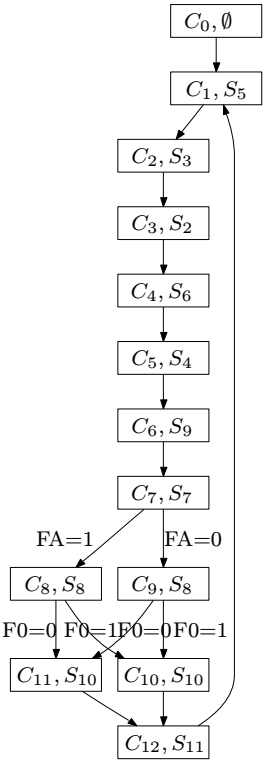


Figure 4 State diagram for the leftmost PE column. The node names C_x, S_y show the selected context x and the associated statement S_y. FA and F0 are the flag results from the ALU operation and the flag register, respectively.

Name	Element	Description
DinMux1	5	1st ALU data input
DinMux2	5	2nd ALU datainput
FinMux1	3	1st ALU flag input
FinMux2	3	2nd ALU flag input
FsMux	9	FSM flag multiplexer
DataReg	7	ALU data write register
DataReg1	7	S input data write register
DataReg2	7	W input data write register
DataReg3	7	E input data write register
FlagReg	8	ALU flag write register
OP	6	ALU operation
MuxN	1	data sent to N output
MuxE	1	data sent to E output
MuxS	1	data sent to S output
MuxW	1	data sent to W output

Table 5 Description of the context memory microcode. Element references the element in Figure 3.

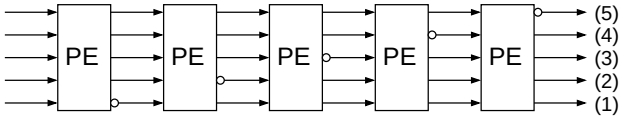
Context	DinMux1	Op	DataReg	FlagReg	DinMux2	FsMux	MuxN	MuxE	MuxS	MuxW	DataReg1	DataReg2	DataReg3
0	N	+	R0	F0	N	N	E	N	N	N	R1	R2	R3
1	R1	+	R2	F0	R0	FA	R0	R4	R3	N	R1	R3	R5
2	S	+	R1	F0	R0	FA	R1	R3	N	N	R5	R3	R5
3	R0	+	R3	F0	R0	FA	E	R3	N	N	R5	R0	R5
4	W	+	R2	F0	R2	FA	R3	R3	N	N	R3	R5	R5
5	R3	+	R1	F0	R1	FA	R2	R0	N	N	R3	R0	R5
6	S	-	R2	F0	R2	FA	R3	R1	W	S	R5	R3	R5
7	W	-	R3	F1	R1	FA	E	R3	W	S	R5	R5	R5
8	N	-	R3	F1	R3	F0	R0	N	N	N	R5	R0	R5
9	N	+	R3	F1	R3	F0	R0	N	N	N	R5	R0	R5
10	N	-	R2	F0	R2	FA	E	R0	N	N	R5	R1	R5
11	N	+	R2	F0	R2	FA	E	R0	N	N	R5	R1	R5
12	R2	+	R4	F0	R3	FA	E	R1	N	N	R5	R1	R5

Table 4 Content of the context memory for the leftmost PE column. The symbols define the configuration for each Mux/FU: N, E, S, W represent the north, east, south and west inputs of the PE and Rx, Fy the corresponding data (x) and flag (y) registers. The FUs (DinMux1, OP, etc.) are summarized in Table 5. The control for the FUs FinMux1, FinMux2, FMuxN, FMuxE, FMuxS, FMuxW is not used.

C_6 the data is read from register 1 of PE $\boldsymbol{\kappa} = \binom{2}{1}$ and communicated over the horizontal channel to $\kappa\binom{3}{1}$, which stores the data in register 3. The data from register 3 is read in C_7 and transfered to $\boldsymbol{\kappa}\binom{4}{1}$ where it is used in the same context to compute $h_3\binom{4}{1}$.

Routing of I/O Data

The specification given in Table 3 also defines the scheduling of I/O data for the leftmost column. All input data in PE row y is routed through the PE $\boldsymbol{\kappa} = \binom{0}{y}$. Hence, the PE receives $p_i(\mathbf{i}+\binom{4}{0})$, $p_i(\mathbf{i}+\binom{3}{0})$, $p_i(\mathbf{i}+\binom{2}{0})$, $p_i(\mathbf{i}+\binom{1}{0})$, $p_i(\mathbf{i})$ on its horizontal channel input. The data is stored in registers of the PE and routed to next PE $\boldsymbol{\kappa} = \binom{1}{y}$ according to the schedule. The specified mapping has an interesting property that makes I/O very efficient: Each PE consumes one input data and produces one output data. It also needs to transfer the input data to all PEs on the right side and the output data from the left side PEs over the horizontal channel inputs and outputs. The mapping specifies a schedule in which, for each PE one input data is replaced by one output data, see Figure 5. The I/O routing is realized such that the effective control is identical for all PEs.



(1) C_8,C_9: $p_i(\mathbf{i}), s(\mathbf{i})$, (2) C_5: $p_i(\mathbf{i}+\binom{1}{0}), s(\mathbf{i}+\binom{1}{0})$,
(3) C_3: $p_i(\mathbf{i}+\binom{2}{0}), s(\mathbf{i}+\binom{2}{0})$, (4) C_2: $p_i(\mathbf{i}+\binom{3}{0}), s(\mathbf{i}+\binom{3}{0})$,
(5) C_1: $p_i(\mathbf{i}+\binom{4}{0}), s(\mathbf{i}+\binom{4}{0})$

Figure 5 I/O Routing for one row of PEs. The arrows indicate data transfers on the same horizontal channel. The origin of output data is at the arrow starting with ∘. For each line, the transfered data and the context is shown below.

5.3 SystemC Simulation of the Processor Array

The SystemC simulation uses an instance of the CRC architecture. It was straightforward to include our proposed extension of the PE control in the model. The PE was used to build a complete array as shown in Figure 6. Since we are using only one horizontal and vertical channel per PE row/column, the interconnect network is simplified. Our Execution scheme uses partitioning and hence, we added local memory to the border of the rows and columns. The horizontal channel uses as inputs either memory content computed in previous partitions or input data of the EDA. Similarly, the horizontal channel output is either stored in local memory or constitutes the EDA output data. The local memory and I/O data control is simulated at the top level, the PA itself executes the PE content and state machine code in a cycle accurate manner. The PA code could be verified and can now be used in Verilog RTL or gate level simulation to evaluate performance, power and area of the implementation.

6 Conclusion

In our case study we have shown that our formal method can be used to derive efficient realizations of parallel algorithms that suit the needs of an independently developed architecture model. The realizations could be verified on the CRC model as target architecture. Our method is implemented in an optimization tool that can be targeted to architectures with varying parameters. Similarly, the CRC model can be instantiated with the parameters used by the optimization tool. This allows a co-exploration of implementation and architectural parameters.
Our optimization tool generates a mapping that is used as a specification to program the CRC model. The SystemC implementation of the CRC model is

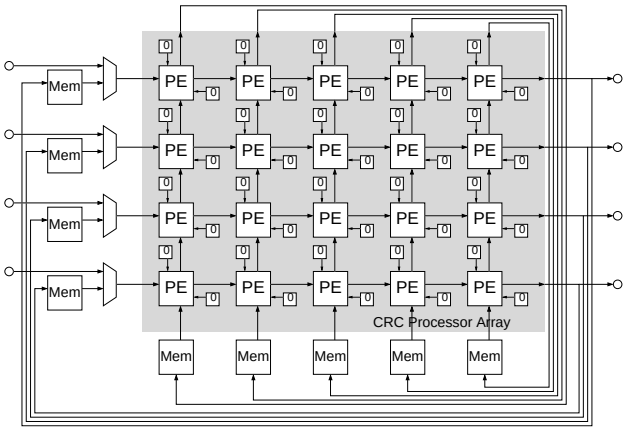


Figure 6 Processor array and periphery for realization (case 1) given in Table 2.

then used for cycle accurate, yet fast, functional simulation of realization. Currently, the designer needs to translate the specification into a CRC program manually. In the future we would like to automate this process. The implementation of the external controller depends on the integration of the CRC model into a SoC, which has not been investigated yet.

We described one realization if the EDA in this paper. The realization has a high throughput despite the simple and cost efficient interconnect network. We have shown how this realization can be scaled to larger processor arrays. It remains a subject of further work to evaluate these realizations on a lower level to gain detailed costs in terms of power/area and clock speed.

7 References

[1] J. A. Brenner, J. C. van der Veen, S. P. Fekete, J. Oliveira Filho, and W. Rosenstiel. Optimal simultaneous scheduling, binding and routing for processor-like reconfigurable architectures. In *16th International Conference on Field Programmable Logic and Applications (FPL)*, 2006.

[2] F. Hannig, H. Dutta, and J. Teich. Mapping of regular nested loop programs to coarse-grained reconfigurable arrays - constraints and methodology. In *18th International Parallel and Distributed Processing Symposium*, page 148, April 2004.

[3] ILOG CPLEX. www.ilog.com/products/cplex.

[4] E. A. L. Charest and A. Tsikhanovich. Designing with systemc: Multi-paradigm modeling and simulation performance evaluation. In *11 International HDL Conference*, 2002.

[5] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures, 2002.

[6] M. Motomura. A dynamically reconfigurable processor architecture. In *Microprocessor Forum*, 2002.

[7] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel. Cost functions for the design of dynamically reconfigurable processor architectures. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004.

[8] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel. A new design approach for processor-like reconfigurable hardware. In *Euro DesignCon*, 2004.

[9] T. Oppold, T. Schweizer, T. Kuhn, W. Rosenstiel, U. Kanus, and W. Straßer. Evaluation of ray casting on processor-like reconfigurable architectures. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2005.

[10] 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.

[11] S. Siegel and R. Merker. Algorithm partitioning including optimized data-reuse for processor arrays. In *Proceedings IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC 2004)*, pages 85–90, September 2004.

[12] S. Siegel and R. Merker. Minimum cost for channels and registers in processor arrays by avoiding redundancy. In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2006)*, pages 28 – 32, Steamboat Springs, USA, September 2006.

[13] S. Siegel and R. Merker. Optimization of communication cost within processor arrays caused by I/O. In *Proceedings of the IASTED 18th International Conference on Parallel and Distributed Computing and Systems (PDCS 2006)*, pages 680–685, November 2006.

[14] SiliconHive. Hivecc datasheet, 2006.

[15] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5):465–481, 2000.