Execution Schemes for Dynamically Reconfigurable Architectures

T. Oppold, T. Schweizer, J. Oliveira Filho, S. Eisenhardt, T. Kuhn, W. Rosenstiel
University of Tuebingen
Wilhelm-Schickard-Institute, Computer Engineering
Sand 14, 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. This paper describes how we establish execution schemes for coarse grained dynamically reconfigurable architectures and presents area, timing, and gate-level power estimations derived from a synthesized architecture model.

I. Introduction

Surveys of reconfigurable computing systems [1][2] show that there is a great diversity of reconfigurable architectures. Likewise, the applications are executed differently on individual architectures with partly common and partly unique features. High-level compilers exist only for a minority of the architectures.

In this work, applications described in the C language and how they can be mapped automatically onto highly reconfigurable architectures are the driving force for identifying architectural resources needed for an efficient execution. By analyzing a variety of real-world examples used in benchmarks ([3][4]) or for visual computing, requirements for the architecture, the compiler, and their interaction are extracted. The results are used to evaluate existing architectures/compilers (e.g., [5] and [6]) with respect to what features are supported and to design new architectures for which a C compiler is readily available since it was co-developed with the architecture.

To provide a flexible target architecture, we have developed the CRC model (Configurable Reconfigurable Core) as a general model for architectures that can be reconfigured in less than a clock cycle. We call such architectures *processor-like reconfigurable architectures.*

One step in the compiler is the scheduling of operations which is usually done after high-level optimizations and transformations. From a scheduled design, an execution scheme can be derived defining for each operation the state and the context in which it is executed. An execution scheme already defines a significant number of architectural features needed to execute it.

Execution schemes are not only an important intermediate format in our design flow but also constitute a useful level to compare architectures. Furthermore, they can guide application developers that use a design entry comparable to the register-transfer-level. From that point of view execution schemes basically describe design patters for reconfigurable computing as described in [7].

In the next section, related work is discussed. In Section III, the architecture model used for this contribution is presented. Section IV describes the process of generating execution schemes by an example. Further examples are discussed in Section V. Section VI provides experimental results for synthesized architecture prototypes.

II. Related Work

Various projects deal with the development of either domain-specific or general-purpose reconfigurable architectures. Targeting coarse grained architectures, in [8] a set of applications representing a domain is analyzed to determine the most appropriate quantity and types of functional units across the domain. Only few works, e.g. [9], focus on power/energy issues. One reason for this is that power estimations require some kind of physical model related to a target technology that can be analyzed. In [10], reconfigurable architectures for general-purpose computing are deeply analyzed including physical implementation, but power/energy is not considered. Today, commercial tools enable power estimations and optimizations at relatively high levels so that such issues can be incorporated in an exploration flow more easily.

A high-level compiler is available only for a minority of the architectures. Even for well established architectures like FPGAs, only a few C-based compilers are at the market (e.g., [11]) and to this day, application design is mostly done with relatively low productivity at the register-transfer-level. This becomes even worse since fine grained FPGAs recently tend to include coarse grained components like dedicated multipliers or block RAM to offer greater computing capabilities. Research on how new architectural features can be utilized efficiently in a high-level design flow (e.g., [12]) is usually done after the devices are available on the market leaving the resources unused in a highly productive design flow for a long time. DRESC is a compiler framework for coarse grained reconfigurable architectures that provides an integrated design flow from C to executable code targeting the ADRES architecture [13].

Similar to the CRC model, ADRES is actually an architecture template that can be modified in an exploration flow. Another similarity is that the architecture explicitly incorporates features (e.g. predicate support) that make it a good fit for the target applications (loops) and the techniques used in the compiler (based on modulo scheduling). Differences to the CRC project include that DRESC uses only unit delay and that the reconfigurable array is tightly coupled to a VLIW processor. In the CRC project, prototypes of architecture instances are synthesized and analyzed frequently to enable compilation techniques from the area of hardware synthesis that require detailed timing information (e.g. operator chaining). This synthesis/analysis is also used to develop and benchmark power optimizing techniques for the compiler and in the architecture. The CRC model can be classified as loosely coupled requiring a dedicated control unit which must be evaluated during architecture exploration as in [10].

Commercial reconfigurable architectures that provide C-based design entry are also on the market [5]. We use such architectures to validate parts of our research. But the commercial nature of these products usually prohibits prototyping new features such as temporal-spatial voltage scaling [14].

III. Architeture Model

The generation of execution schemes requires a model of the target architecture. For this purpose, the CRC model is used as a model for processor-like reconfigurable architectures. It consists of an array of processing elements (PE), a reconfigurable interconnect network, and memory. Each PE consists of a functional unit (FU) that is able to perform arithmetic and logic operations, a register set, and configuration memory. By the configuration memory, multiple contexts for execution can be defined. By means of processor-like reconfiguration, the context can be switched on a cycle-by-cycle basis. Since the CRC model is firstly a theoretical model these resources are not constrained. The model can be configured with a variety of parameters to define instances of the CRC model. These instances are real architectures with limited resources.

Fig. 1 shows the PE of a possible instance of the CRC model that implements a nearest neighbor interconnect network. The configuration memory is subdivided into context memory and memory in a control unit that stores state transitions to implement a finite state machine (FSM). Both memories are configured from outside when the device is booted.

The output of the FSM selects an entry in the context memory. The output of the context memory selects the operands for the FU. The operands can come from the ports north, east, south, or west (N, E, S, W) or from internal registers of the PE. Ports can be connected to a neighboring PE or they can be ports of the device if the PE is located at the border of the array. The context memory also determines the operation performed by the FU and destination registers for the data and 1-bit status outputs of the FU if the result is to be stored in a register of the PE. The

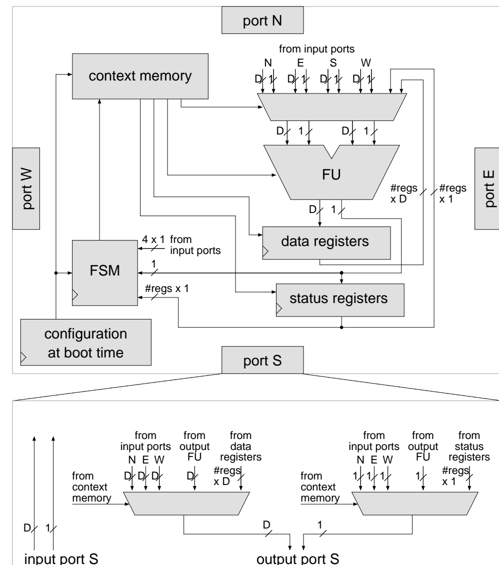

Fig. 1. A PE of a possible architecture instance. D denotes the width of the data path that can very for different implementations.

status output is used for carry signals and for the results of compare operations. To determine the next state, the status output of the FU, the output of the status registers, and the status signals of the four ports are connected to the FSM. The FSM and the register are synchronized by a common clock.

The lower part of Fig. 1 shows the port S module of the PE in detail; the port N, E, and W modules are equivalent. The data and status input is routed directly to the inside of the PE.

The context memory determines which data and status signals are available at the output ports. This is done independently for data and status signals. The source for the output ports can be the output of the FU, the output of any of the registers, and the input port of any of the other ports. The output of the FU is selected to realize operator chaining. The output of one of the other ports are used to implement the nearest neighbor interconnect network. A PE is able to simultaneously execute an operation in the FU and to route data and status signals from one neighbor to another.

Instances of the CRC model are described in Verilog so that they can be synthesized and analyzed to obtain area and timing estimations. After mapping an application onto an architecture instance, the execution is simulated to verify the implementation and to obtain power estimations based on the implementation's switching activity.

IV. Generation of Execution Schemes

The flow for generating execution schemes is depicted in Fig. 2. After a brief summary of the different steps in the process, details are presented by an example.

The application is analyzed and the operations and data types that are used are extracted. An architecture proto-

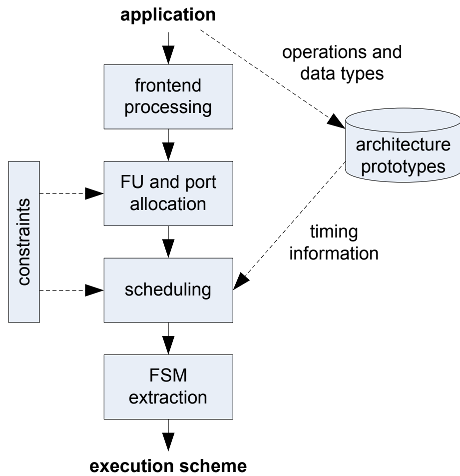

Fig. 2. Flow for generating execution schemes.

type that fits these requirements is selected from a database or a new prototype is created if necessary. The timing information of this prototype is required for the scheduling phase. Since the execution schemes are used to explore various architecture alternatives in an early stage, the timing information can only be used as a first estimation. After an execution schemes is established and a suitable architecture instance is defined, timing information is extracted for this instance and then the actual clock speed can be estimated for this target architecture.

The allocation determines the minimal number of FUs and I/O ports required to meet a given performance constraints. This performance constrained approach reflects a system designers point of view. A system designer typically fixes the needed bandwidth or the available memory of a system or varies it only within tight limits. After that, a component that provides the desired performance at the lowest costs or power requirements is chosen. It must be noted that execution schemes are only an intermediate representation and that additional resources like interconnections need to be allocated in later steps. An optimal execution may therefore eventually result in a suboptimal implementation.

For the scheduling of operations, well known techniques like ASAP or list scheduling are employed. Additionally, scheduling or parts thereof is done manually using deep knowledge about the application. By doing so, advanced techniques are identified for future implementation in the compiler.

To control the reconfiguration of the PEs during runtime, a state machine is extracted that can be mapped to the control unit of the CRC model.

A. Application

To exemplify the basic flow, the luminance calculation of the RGB to YIQ conversion from the EEMBC (Embedded Microprocessor Benchmark Consortium) Consumer Benchmark [3] will be used in the following and RGB2Y will be used to refer to this example.

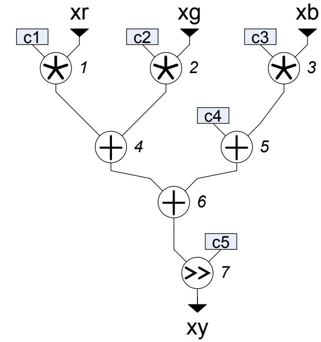

Fig. 3. CDFG for the RGB2Y-example.

RGB to YIQ conversion is used in the NTSC encoder where the RGB inputs from the camera are converted to a luminance (Y) and two chrominance information (I, Q). The luminance is defined by the following equation:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

The original benchmark includes the calculation of I and Q as well as address calculation for reading the RGB data from memory in a loop over all image pixels. For the following evaluation, it is assumed that the RGB data is provided at the input ports of the array by a previous stage in the surrounding environment.

Since the EEMBC benchmarks are tailored for embedded processors, the source code provides a fix point implementation for the above equation and specifies the data types:

$$xy = (c1 * xr + c2 * xg + c3 * xb + c4) >> c5;$$

The result (xy) and the inputs (xr, xg, and xb) are 8-bit values. The constants $c1$ to $c4$ are 16-bit values and $c5$ is a 4-bit value. All values are unsigned.

For the generation of execution schemes it is assumed that each operation can be mapped directly to an FU. Therefore, an architecture that supports 16x16-bit multiplication, 32-bit addition, and shift right by 16 bits must be selected.

B. Frontend Processing

From the application description, a combined control and data flow graph (CDFG) is derived using a commercial compiler framework [15] that is able to perform high-level transformations/optimizations like loop unrolling or dead code elimination. Applying such transformations results in various CDFGs derived from a single application description.

Fig. 3 depicts the CDFG that is derived from the source code after transformation into static single assignment form (SSA). SSA is an intermediate representation in which every variable is assigned only once. This is useful for high-level optimizations and exhibits instruction level parallelism.

C. Constraints

To generate reasonable execution schemes, it is necessary to define realistic constraints. To optimize for minimal area

without performance constraints is not helpful in our approach. It would always lead to an execution scheme using only 1 PE. To optimize for maximum performance without any constraints is also not helpful. Applications that are good candidates for an execution on a reconfigurable array very often operate on a continuous data stream or at least on a large amount of data like digital images with millions of pixels. Theoretically, almost arbitrary performance can be achieved if such applications can be parallelized without limitations.

A constraint that is well comprehensible is the rate at which the data is processed by a reconfigurable array in an embedded system. This rate can be appointed by previous or subsequent stages in the processing of data streams or by the availability of memories.

In the context of pipelined execution, the term *Initialization Interval* (II) is commonly used in the literature. In the following, the II is used analogically to specify the number of clock cycles that are available to consume a set of input values IN. For the example application, the set of input values is defined as follows:

$$IN = \{xr, xg, xb\}$$

An application might also be constrained by the output rather than by the input. In the following, it is assumed that input requirements dominate.

As an extension to the II, the *Data Input Rate* (DIR) is used to specify the number of input values along with the II at which they are consumed:

$$DIR = |IN| \,/\, II$$

For the example application, reading IN every clock cycle ($DIR = 3/1$) and reading IN within 3 clock cycles ($DIR = 3/3$) is evaluated.

Unlike many other coarse grained reconfigurable architectures, the CRC model allows it to chain two or more data-dependent operations within one clock cycle. To explore this degree of freedom, a target clock speed must be set that constrains the number of operations in a chain. To specify this clock speed constraint, DIR_f is defined as the DIR multiplied by the clock frequency:

$$DIR_f = DIR * clock\,frequency$$

Since no real constraints for the clock speed are available for the generation of execution schemes, the clock period is set to values that exhibit interesting execution schemes. From the timing information of the used architecture prototype it can be anticipated that a clock period constraint of 5 ns will inhibit operator chaining, while a clock period of 13 ns will allow it to chain all operators of the example application.

In the following evaluation, execution schemes for $DIR_f = (3/1) * 77$ MHz , $DIR_f = (3/1) * 200$ MHz, and $DIR_f = (3/3) * 200$ MHz will be analyzed for the example application.

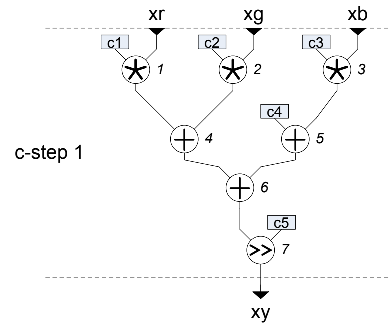

Fig. 4. Schedule for the RGB2Y-example ($DIR_f = (3/1) * 77$ MHz).

D. FU and Port Allocation

Allocation is defined as the process of deciding how many and which kind of resources can be used for the implementation. Currently, we consider only homogeneous arrays with identical PEs. This reduces the FU allocation task to deciding how many FUs should be used.

Processor-like reconfiguration allows it to reuse the FUs for different operations in different clock cycles. Let OP be the set of operations in the CDFG. For a CDFG with no branches in the control flow, the lower bound for the number of required FUs can be derived directly from the number of operations in the CDFG and the II:

$$\#FUs = \left\lceil \frac{|OP|}{II} \right\rceil$$

The CDFG of the example comprises 7 operations, hence 7 FUs are required for $DIR = 3/1$. For $DIR = 3/3$, the 7 operations can be distributed over 3 clock cycles and therefore at least 3 FUs are required.

Although a physical port itself is not considered as expensive in terms of area or power, it can represent a large memory block with significant costs. The lower bound for the number of physical input ports can be derived directly from the DIR:

$$\#inports = \lceil DIR \rceil$$

The lower bounds for the number of FUs and input ports are used as additional constraints in the scheduling phase since adding more of these resources can not improve the performance.

E. Scheduling

During the scheduling process each operation is assigned to a control step (c-step) that will be executed within one clock cycle by the target architecture. The timing information that is needed to perform the scheduling is taken from an existing instance of the CRC model that fits the requirements of the example application. Detailed timing information for this prototype is published in [16].

Fig. 4 shows the schedule for $DIR_f = (3/1) * 77$ MHz. All input values are read in one single c-step and processed in that c-step.

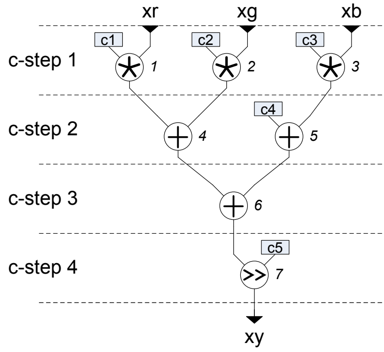

Fig. 5. Schedule for the RGB2Y-example ($DIR_f = (3/1) * 200$ MHz).

The schedule for $DIR_f = (3/1) * 200$ MHz is shown in Fig. 5. The clock period constraint requires it to schedule the operations for execution in 4 different c-steps without operator chaining. But if the 4 c-steps are executed sequentially, the II constraint is violated. This dilemma is solved by pipelining the execution. To achieve this, c-steps 2, 3, and 4 are reassigned to c-step 1. By doing so, all operations are scheduled for execution in the same clock cycle without violating the constraints set by the DIR_f and the preceding resource allocation. At this step in the process, no further action is taken for pipelining.

For the previous two DIR_f constraints, all scheduling algorithms currently implemented in our compiler (ASAP (As Soon As Possible), ALAP (As Late As Possible), and list scheduling) yield the same schedule. For $DIR_f = (3/3) * 200$ MHz, slightly different results are generated by each algorithm. The result of the ASAP scheduling algorithm is shown in Fig. 6. The 3 input values are read in the first 3 c-steps, i.e., as soon as the physical input port is available. To satisfy the II constraint, the 6 c-steps must be mapped to 3 clock cycles, resulting in a pipelined execution. This is done by reassigning all c-steps $n > II$ to c-step $(n \bmod II)$.

For the example application, pipelining works even with simple ASAP scheduling. But in the general case ASAP is not suitable since it does not consider resource constraints. To constrain the number of FUs used in a c-step, list scheduling can be used. To enable pipelining without violating the $\#FU$ constraint, the list scheduling algorithm must be extended so that the $\#FU$ constraint is not violated for one clock cycle. This is achieved by counting the number of used FUs not only for one single c-step n during scheduling but for all c-steps $(n \bmod II)$ together since all these c-steps will be executed in one clock cycle after pipelining is applied.

F. FSM Extraction

The c-steps defined by the scheduling must be executed repeatedly one after the other. To control this sequential behavior, an FSM is constructed.

The FSMs for $DIR_f = (3/1) * 77$ MHz and $DIR_f = (3/1) * 200$ MHz are trivial since there is only one c-step after

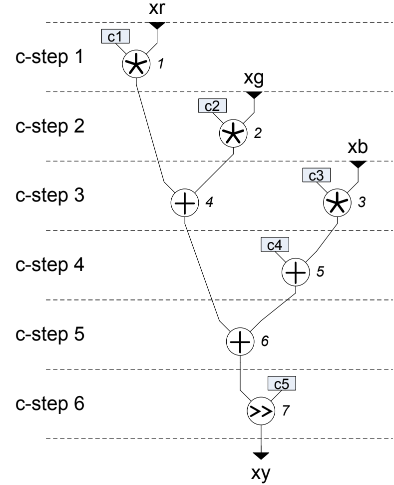

Fig. 6. Schedule for the RGB2Y-example ($DIR_f = (3/3) * 200$ MHz).

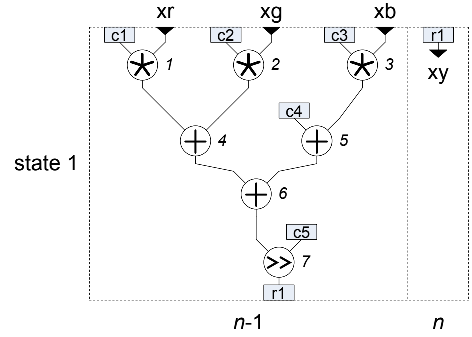

Fig. 7. States of the RGB2Y-example for $DIR_f = (3/1) * 77$ MHz.

scheduling/pipelining. As depicted in Fig. 7 and Fig. 8, all operations are mapped to state 1 and hence no state transitions must be considered. The states are horizontally subdivided by dotted lines and annotated underneath with the set of input values that is processed by that subdivision in a given clock cycle.

The figures depicting the states also include the registers that are required at the borders of clock cycles. Although register allocation is actually done in a later step based on lifetime analysis, the results of a preliminary manual allocation are included in the figures to allow tracing the data flow over multiple clock cycles. For $DIR_f = (3/1) * 200$ MHz, the outputs of all FUs are stored in registers to enable pipelined execution. For $DIR_f = (3/1) * 77$ MHz,

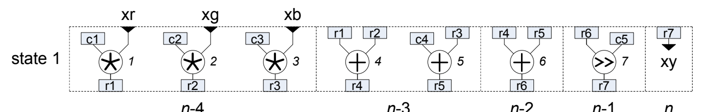

Fig. 8. States of the RGB2Y-example for $DIR_f = (3/1) * 200$ MHz.

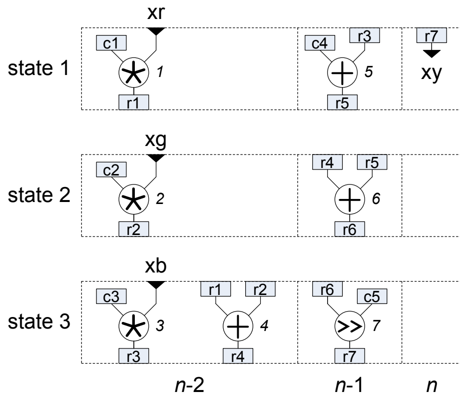

Fig. 9. States of the RGB2Y-example for $DIR_f = (3/3) * 200$ MHz.

only the result is stored in a register. The register for the result is necessary because it is assumed that results are read by the surrounding environment in the following clock cycle.

For $DIR_f = (3/3) * 200$ MHz, the schedule comprises 3 c-steps after pipelining. The FSM must be able to generate a sequence of states $1 \rightarrow 2 \rightarrow 3$ and then start with state 1 again. Fig. 9 shows the operations that are executed in each of the states.

Besides the state transitions, the context that is selected in each state must be specified to characterize the required FSM. The context of a PE determines the operation performed by the FU as well as the behavior of other resources that are subject to later steps in the compiler. The output of the FSM must select a context for all PEs so that the PEs together implement that part of the data path that is needed in the current state.

For $DIR_f = (3/1) * 200$ MHz and $DIR_f = (3/1) * 77$ MHz, no state transitions occur and therefore a single context can be assigned statically for all PEs.

For $DIR_f = (3/3) * 200$ MHz, the context of all PEs can be selected directly by the state. As special case of a Moore machine where the output is directly determined by the state, a Medvedev machine is sufficient to handle the control.

The PE described in section III features a dedicated control unit to implement an FSM. As an alternative solution, groups of PEs or the entire array could be controlled by one common unit. The FSM for $DIR_f = (3/3) * 200$ MHz can be implemented easily regardless of the control unit being dedicated to one or to many PEs.

V. Examples

In this section, the execution schemes for the RGB2Y-example are summarized and two further examples are discussed. Table I provides an overview of the execution schemes for all examples. It specifies the high-level transformations applied during frontend processing, the DIR_f constraint, and the resulting throughput as well as the latency for processing a single input sample. The resource requirements are given as the number of required input ports,

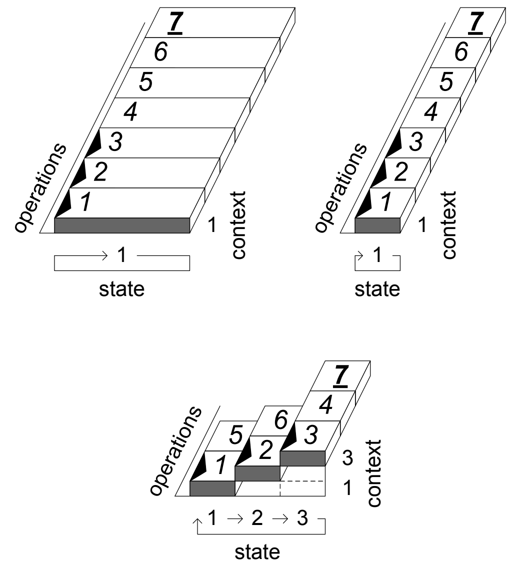

Fig. 10. Execution schemes for the RGB2Y-example (top-left: $DIR_f = (3/1) * 77$ MHz, top-right: $DIR_f = (3/1) * 200$ MHz, bottom: $DIR_f = (3/3) * 200$ MHz).

FUs, states, and contexts. For the FSM, it is specified how many independent state machines are required and which type is necessary.

A. Luminance Calculation of the RGB to YIQ Conversion

Fig. 10 shows 3D-representations of the execution schemes for the RGB2Y-example with focus on the resource usage. The extent of the grey boxes along the state-axis indicates the time spent in a state, i.e., the clock period. The context-axis denotes the context that is selected by a state. On the operations-axis, the usage of FUs is shown for each context. The operation that outputs the result is underlined. The inputs are labeled by black triangles.

B. High Pass Grey-Scale Filter

As a second example, the high pass grey-scale filter from the EEMBC Consumer Benchmark is evaluated. This 2D-filter is used in the front end processing of digital cameras to sharpen images. Similar to the RGB to YIQ conversion, a sum of products is calculated from 9 input values and shifted by a constant value:

$$out = (c1 * in1 + c2 * in2 + ... + c9 * in9) >> c10.$$

The set of input values $IN = \{in1, in2, ..., in9\}$ constitutes a 3x3 neighborhood of pixels with $in5$ being the center pixel that is to be recalculated. The original benchmark includes the address calculation for reading the image data from memory and implements a subsequent stage to black out the borders of the output image. As in the previous example, only the above calculation is considered for the evaluation and is referred to as HPGS.

Using the techniques described in the previous section, similar execution schemes can be generated for the HPGS-

TABLE I
Summary of the Execution Schemes

application	frontend	DIR_f	samples/s	latency	inports	FUs	states	contexts	FSM
RGB2Y	SSA	$(3/1) * 77$ MHz	77,000,000	13 ns	3	7	1	1	none
RGB2Y	SSA	$(3/1) * 200$ MHz	200,000,000	20 ns	3	7	1	1	none
RGB2Y	SSA	$(3/3) * 200$ MHz	66,666,667	30 ns	1	3	3	3	1 Medv.
HPGS	SSA	$(9/18) * 200$ MHz	11,111,111	30 ns	1	1	18	18	1 Medv.
HPGS	none	$(9/18) * 200$ MHz	11,111,111	90 ns	1	1	18	11	1 Moore
resampling	n/a	$(8/1) * 200$ MHz	≤200,000,000	≥60 ns	8	28	2	2	1 Medv.
resampling	n/a	$(8/1) * 200$ MHz	200,000,000	60 ns	8	28	2	2	12 Medv.

example. In the following, two execution schemes that exhibit new requirements for the FSM are discussed. For $DIR_f = (9/18) * 200$ MHz, all 18 operations of the application can be mapped to a single FU. If SSA transformation is done in the frontend processing, a CDFG similar to the one depicted in Fig. 3 is generated resulting in a latency of 6 clock cycles. If no SSA transformation is applied, a CDFG with a rather sequential characteristic is generated resulting in a latency of 18 clock cycles. The regular structure of this CDFG allows a register allocation so that all add-operations read the same input registers and also write to the same output register. Since no other functionality must be provided in the states performing the add-operations, this specifies the required context exhaustively. Hence, multiple states can be mapped to a single context reducing the required contexts from 18 to 11. This many-to-one mapping can not be handled by a Medvedev machine but requires a Moore machine to be implemented by the control unit.

C. Resampling Stage of the Ray Casting Algorithm

Ray casting is a real-life application for the visualization of 3D scientific and medical data with high performance requirements. The ray casting algorithm is usually implemented in a pipeline of five major stages. Resampling is the most computational intensive stage of those five stages and is done by different filter kernels. Based on the results of previous stages, one of the filters is chosen dynamically during operation resulting in a CDFG with corresponding branches in the control flow. An execution scheme for the resampling stage that reduces the number of FUs without decreasing performance keeps different filters in the context memory and switches between them as they are needed. Details on executing ray casting on processor-like reconfigurable hardware are presented in [17]. In the following, only the FSM for $DIR_f = (8/1) * 200$ MHz is discussed.

The resampling stage processes 8 input values. Commonly used filters for resampling are trilinear interpolation and nearest neighbor interpolation. The results in Table I are based on implementing these two filter kernels. To meet the $DIR_f = (8/1) * 200$ MHz constraint, resampling must be executed in a pipeline. The implementation of trilinear interpolation contains 28 operations that can be distributed over 12 pipeline stages while nearest neighbor interpolation requires fewer resources. Therefore, the total number of required FUs and the required pipeline stages are determined only by trilinear interpolation. If there is only one FSM for all PEs, the pipeline must be flushed before the filter kernel can be changed resulting in reduced overall throughput and increased latency. If a dedicated FSM for each of the stages is provided, the throughput can be sustained at the maximum rate.

TABLE II
Summary of the RGB2Y-example running on instances of the CRC model

DIR_f	PEs	clk [MHz]	exe [μs]	$power$ [mW]	$energy$ [nJ]
$(3/1) * 77$ MHz	3x3	100	1.0	36.6	36.6
$(3/1) * 200$ MHz	3x3	200	0.5	44.5	22.2
$(3/3) * 200$ MHz	1x3	200	1.5	25.0	37.4

VI. Experimental Results

After an execution scheme is established, the next step is the binding of the operations and registers to the components of an array of PEs. This binding as well as establishing the necessary interconnections is currently done graphically by hand. It is straightforward to derive an assembler-like symbolic description of the behavior of a PE in each context from the graphical representation. The assembler-like description is translated automatically into a binary format that can be written directly to the context memory. The state transitions for the FSM, as well as the output functions in the case of a Moore machine, can be coded easily by hand so that they can be written directly to the memory of the control unit.

The results of mapping the three execution schemes presented for the RGB2Y-example and executing them on two different instances of the CRC model is summarized in Table II. Both instances feature the same PEs as described in Section III and further specified for the RGB2Y-example in Section IV-A. Each PE provides 12 data registers, 8 contexts, and 8 states for the FSM.

For the synthesis of the instances, a 130 nm standard cell

library was used as the target technology. The estimated area for the 3x3 array is $742,870\ \mu m^2$, and $247,623\ \mu m^2$ for the 1x3 array of PEs. The actually achieved clock speed (*clk* in the table) for the execution scheme that uses operator chaining is 100 MHz instead of the 77 MHz initially estimated.

The execution time (*exe*) and energy consumption (*energy*) in the table are given for processing 100 samples of an input image. The power dissipation (*power*) is estimated by a commercial analysis tool after simulating the execution at the gate-level and recording the switching activity.

The best results in terms of performance and energy consumption are achieved by executing the application in a pipeline without reconfiguration. Using processor-like reconfiguration yields the lowest area and power dissipation. Operator chaining reduces the latency for processing a single input sample but decreases throughput and does not improve energy consumption compared to pipelining. The experimental results suggest that for $DIR = (3/1) * 77$ MHz, execution in a pipeline at this clock speed is a better choice than chaining the operations. This would also yield the best results in terms of power dissipation and if the supply voltage can be reduced, power dissipation and energy consumption can be further improved without violating the performance constraints.

VII. Conclusions and Further Work

The presented execution schemes show that the number of FUs can be reduced significantly by processor-like reconfiguration if the control flow of the application or the performance constrains allow it. Since doubling the number of contexts in one PE requires less area than duplicating the entire PE, reduced manufacturing costs can be expected for such cases. On the other hand, processor-like reconfiguration requires additional resources compared to a statically reconfigurable architecture and therefore affects area, performance, and power dissipation of one PE adversely. Ongoing work in the CRC project includes detailed comparisons to fine and coarse grained statically reconfigurable architectures as well as to ASIC solutions to asses the costs imposed by reconfigurability.

The presented experimental results show that pipelining the execution yields the best results for the example application if sufficient resources are provided by the target architecture. Since the lowest latency for processing one input sample is achieved by operator chaining, this technique will be further considered for applications that can not be executed in a pipeline.

The architecture prototypes used for the experimental results provide significantly more resources than actually used by the application. This is a common situation for coarse and fine grained reconfigurable architectures. Future work in the project will focus on application domains so that for each domain an architecture with minimal resource or power requirements and yet sufficient flexibility and performance is defined.

Acknowledgements

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

References

[1] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[2] F-J. Veredas-Ramirez, M. Scheppler, and H-J. Pfleiderer. A survey on reconfigurable computing systems: Taxonomy and metrics. In *IV Workshop on Reconfigurable Computing and Applications (JCRA)*, Spain, 2004.

[3] Embedded Microprocessor Benchmark Consortium (EEMBC). http://www.eembc.org.

[4] Benchmarks for the 1992 High Level Synthesis Workshop. http://ftp.ics.uci.edu/pub/hlsynth/HLSynth92.

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

[6] V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2001.

[7] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[8] S. Hauck K. Eguro. Resource allocation for coarse grain FPGA development. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 2005.

[9] A. Abnous and J. Rabaey. Ultra-low-power domain-specific multimedia processors. In *IEEE VLSI Signal Processing Workshop*, 1996.

[10] A. DeHon. Reconfigurable architectures for general-purpose computing. Technical Report AITR-1586, MIT Artificial Intelligence Laboratory, September 2 1996.

[11] Celoxica. http://www.celoxica.com.

[12] R. Kastner W. Gong, G. Wang. Storage assignment during high-level synthesis for configurable architectures. In *International Conference on Computer Aided Design (ICCAD)*, 2005.

[13] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Design, Automation and Test in Europe (DATE)*, 2004.

[14] 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.

[15] LANCE Retargetable C Compiler. http://www.lancecompiler.com.

[16] 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.

[17] 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.