

# Cost Functions for the Design of Dynamically Reconfigurable Processor Architectures

Tobias Oppold, Thomas Schweizer, Tommy Kuhn, Wolfgang Rosenstiel  
University of Tuebingen  
Wilhelm-Schickard-Institute, Computer Engineering  
Sand 14, 72076 Tuebingen, Germany  
crc@informatik.uni-tuebingen.de

**Abstract—** There are a growing number of reconfigurable architectures that combine the advantages of a hardwired implementation (performance, power consumption) with the advantages of a software solution (flexibility, time to market). Today, there are devices on the market that can be dynamically reconfigured at run-time within one clock cycle. But the benefits of these architectures can only be utilized if applications can be mapped efficiently. In this paper we describe a design approach for reconfigurable architectures that takes into account the three aspects architecture, compiler, and applications. To realize the proposed design flow we developed a synthesizable architecture model. From this model we obtain estimations for speed, area, and power that are used to provide the compiler with the necessary timing information and to optimize the architecture.

## I. INTRODUCTION

A lot of research in the area of reconfigurable computing systems is focused on the re-use of devices like FPGAs for different applications. But compared to this, newly developed devices provide much more advantages. They allow re-using the functional elements of a device for different operations within a single application. This is accomplished by extremely fast reconfiguration (in less than a clock cycle) during run-time. For these devices, frequent reconfiguration is part of the regular execution. The reconfiguration keeping pace with the execution yields an additional degree of freedom that constitutes a new principle of reconfiguration. We name this new principle processor-like reconfiguration. By means of reconfigurable data paths, processor-like reconfiguration allows it to instantiate and to execute within one clock cycle exactly that part of a circuit that is needed in this cycle. Although for each of the new architectures distinct advantages have been presented, by today none of them was able to gain significant market shares in industry. From the technical point of view, our practical experiences have shown the following reasons for this: 1) the design tool to map applications onto the architecture (the compiler) is developed after the architecture was defined, 2) the language provided to users for application design is not appropriate, or 3) suitable applications are sought after the architecture was developed. This demonstrates that a good

architecture by itself does not provide a good solution. In fact, the benefits of reconfigurability can only be exploited if applications can be mapped efficiently onto the available architectures.

In this context we focus on two problems that need to be solved. The first problem is that different applications need different architectures. This is due to the fact that the area of *Reconfigurable Computing* mostly stresses the use of coarse grained architectures [1]. Accordingly, the commercial architectures are also coarse grained. For such architectures various properties (e.g. the data path width) are fixed at fabrication time of the device. This means that only applications that fit these properties can be mapped onto these architectures without wasting a significant amount of resources. The second problem that we focus on is the availability of a compiler to map applications onto the architecture. Such a compiler must be able to automatically map applications that are described on high levels of abstraction, i.e., the algorithmic and system level. In our opinion application design at the register-transfer (RT) level is not an option. In ASIC/FPGA design the implementation at the RT level already leads to a productivity gap between the number of gates provided by modern process technologies and the number of gates that can be implemented by application design teams. Reconfiguration yields a further degree of freedom that increases design complexity even more.

Our approach to solve these problems is a design environment for reconfigurable architectures that takes into account the three aspects architecture, compiler, and applications. The environment supports an iterative flow where the architecture and the compiler are refined concurrently. This refinement is driven by the analysis of applications from a given problem domain (e.g. encryption or graphical systems). The result of the iterative flow is an architecture together with a compiler which are optimized for the specified problem domain.

As the starting point for the architecture we have defined the CRC model depicted in Fig. 1. The CRC model is a very general model for processor-like reconfigurable architectures that can be configured with a variety of parameters during the refinement process. This configuration and the reconfiguration of the architecture during run-time account for the name CRC (Configurable Reconfigurable Core). It is oriented towards commercially available architectures to

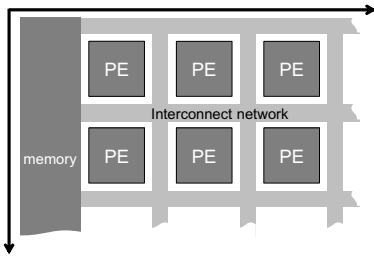


Fig. 1. CRC model.

take into account the technological feasibility but it leaves sufficient space for optimization by abstracting from the real architectures. The CRC model consists of processing elements (PE), an 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 a context memory. An entry in the context memory determines the context of the current clock cycle. The context determines the operation performed by the FU, a register from the register set, and the connections between the PEs. 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. Within the design environment instances of the CRC model are created. These instances are real architectures with limited resources.

In order to take advantage of reconfigurability a proper temporal partitioning of an application must be found. Research activities in the last years have shown that such a partitioning is hard to find if reconfiguration is expensive in terms of time, area, or power consumption. The properties of processor-like reconfigurable architectures allow us to make use of the well known techniques from C-based synthesis. To gain acceptance in industry the compiler must support an input language that is well adopted. Today, C is widely used for algorithm and embedded system design. Hence, we use C as the language for application design in our approach and consider application domains where compute intensive parts of an application are moved from software to hardware to meet performance or power constraints. Object oriented languages like C++ and Java are extensions of C and therefore we are well prepared for a migration to object oriented design that is part of our research on reconfigurable architectures [2].

To realize the flow within our design environment it is necessary to have detailed knowledge about the properties of the architecture. Cost functions are needed to assess the results of the refinement process and to rate possible modifications of the compiler or the architecture. We obtain these cost functions from synthesizing instances of the CRC model. The timing information from the synthesis is also an important prerequisite to perform the application mapping. Scheduling algorithms known from C-based synthesis consider operator chaining to optimize the timing of a design. While superscalar microprocessors like VLIW processors can not make use of this technique the CRC model does support operator chaining. To apply operator

chaining reasonably the timing information for the different operations must be available.

The paper is organized as follows. In the next section we present work that is related to our project. The design environment is described in section III. Section IV introduces a set of instances of the CRC model. Speed, area, and power estimations for our architecture model are presented in section V. Section VI concludes this paper and provides an outlook on further work in the project.

## II. RELATED WORK

We see our design environment as an addition to the work done by industry and other academic research groups. We integrate features of a wide range of reconfigurable architectures and benchmark and compare these features. The results are used in our design flow to optimize existing architectures and to propose new architectural features.

For instance, NEC's DRP architecture is closely related to our architecture model. The DRP was architected based on a clear picture of how C code is compiled into hardware [3] and accordingly the compiler is based on NEC's C-based behavior synthesizer Cyber [4]. Other commercial architectures that influence our architecture model include PACT XPP [5], IPFlex DAP/DNA [6], PicoChip's picoArray [7], and Quicksilver's ACM [8]. Besides these relatively new architectures that are usually labeled as (dynamically) reconfigurable processors we also consider properties of traditional architectures like FPGAs, DSPs, and VLIW processors.

In the academic field, DeHon has done some fundamental work on optimizing area and performance of reconfigurable architectures [9]. He proposed a variety of highly reconfigurable architectures including DPGA and MATRIX that are very valuable for our approach. Morphosys [10] is another architecture that is related to our architecture model. An overview of reconfigurable devices that more or less relate to our work can be found, e.g., in [11].

All architectures mentioned above strongly influence the design of our CRC model. But we do not solely assess the architecture. We also take into account the compiler and the applications that are to be mapped onto the architecture. The concurrent refinement of architecture and compiler plays a prominent role in our approach. In that respect the Teramac project [12] is similar. The project targets logic simulation and to avoid compilation times of many days the architecture was designed according to the needs of the compiler. Teramac does not consider fast reconfiguration during run-time though.

Besides the usually well researched optimization of performance and area, power consumption is a major issue in many of present and future designs. Reconfigurable devices have the potential to provide an excellent performance to power ratio and power dissipation is an important criterion in our design environment. In [13], for instance, techniques to optimize power consumption of reconfigurable architectures are proposed. The commercial tools available today for hardware design consider power consumption mostly at relatively low levels. In [14], power optimization techniques

are considered at high levels of abstraction in hardware design. For microprocessors power consumption is also optimized mostly at the architectural level and compiler techniques are rather in the realm of academic research. In our design flow we consider power at the architectural level as well as in the compiler.

### III. DESIGN ENVIRONMENT

Within the design environment we not only explore different architectures but we vary the applications and the compiler as well. This leads to a very complex design space that needs to be searched. Of course, it would be desirable to have a system that reads in a set of applications and automatically generates an architecture along with a compiler that are optimized for the given applications. Given that processor-like reconfigurable architectures are relatively new and it is not clear yet, for example, what is the best way to map applications or what are the right application domains we see this as an unrealistic goal for the near future. We therefore require user interaction in the design flow. Especially in the early phase of the project the parameters by which the CRC model can be configured are not only scalar values like, e.g., the number of contexts or the width of the data path. In fact, the design team of the compiler reports bottlenecks or proposes improvements of the architecture and this can require a new architecture model that incorporates entirely new features. For instance, if the compiler is not able to map loops efficiently due to missing wiring resources a new interconnect network with fast feedback wires is created.

Fig. 2 depicts the flow in the design environment. The first step is to select an application from an application domain. We mainly consider applications in the area of mobile communication devices and graphical interactive systems. In these areas compute intensive applications can be found that are good candidates for an implementation on reconfigurable architectures. In a first approach we define application domains within these areas by the following criteria: data path versus control oriented applications, applications that benefit from specialized operations (e.g. saturated arithmetic), and single-threaded versus multi-threaded applications. Support of multi-threaded applications is important for system level design and scheduling of multiple threads is a very good match for processor-like reconfiguration since threads can be scheduled on the basis of clock cycles with very little overhead.

The selected application is at first translated into an architecture independent format. This format unveils basic information about the application like the data types used or the memory accesses. Based on that information we manually select an instance of the CRC model that fits the application already to some extent because it provides the right data path width etc. The delays of the various components of this particular instance that are needed for C-based application mapping as well as an area estimation are obtained from synthesizing the instance with a commercial tool. To avoid unnecessary synthesis runs we maintain a database with the results of all synthesized instances.

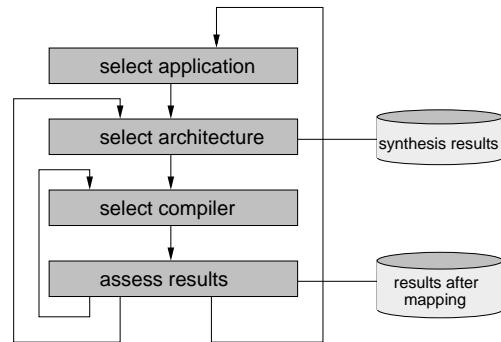


Fig. 2. Flow in the design environment.

In the next step the application is mapped onto the architecture. This is done by selecting an instance of the compiler. An instance of the compiler is defined by the optimizations and compile strategies that are applied during the application mapping. Exploring different techniques in the compiler is a key task in our research. Therefore, the selection of an instance of the compiler often involves implementing new functions in the compiler. These functions are available afterwards as compiler options that can be tried in different combinations. The mapping process is based on well known techniques from C-based synthesis. C-based synthesis transforms an algorithmic description into a temporally partitioned data path and a control unit. The different partitions of the data path can be mapped to different contexts and each of the contexts is able to implement exactly that part of the data path that is needed in this specific clock cycle in an ASIC-like fashion. The compiler that we currently develop is based on our synthesis tool CADDY II [15]. Further information about the techniques that we use in the compiler can be found in [16].

After having performed the application mapping it is known at what clock speed the application can run and how many clock cycles are needed to execute it. The results for all properties (energy consumption, area, and performance) of the application mapped by this specific compile run onto this specific instance of the CRC model are kept in a database for comparison of the results. At this stage in the flow we also have detailed information about the utilization of the architecture's resources. Based on this information we try other compile strategies as well as other instances of the CRC model in order to optimize one of the properties while meeting constraints for the remaining properties.

After an architecture-compiler pair is found that satisfies the requirements for this specific application another application from the application domain is selected and the same steps are performed for this application. This time the initial selection of the architecture can be performed based on architectures found in previous iterations. After having processed a number of representative applications of a domain we assess the flexibility of each of the architecture-compiler pairs by mapping all applications again using this specific pair. If there is an architecture-compiler pair that

is able to meet the constraints for all the applications this is the final result of the flow. If such a pair was not found new architecture-compiler pairs can be tried. If this also fails because the requirements of the applications are too diverse the application domain must be redefined.

#### IV. ARCHITECTURE INSTANCES

As mentioned in the previous section the compiler generates a temporally partitioned data path and a control unit. While the different partitions of the data path can be implemented by the FUs inside the PEs of the CRC model efficiently, this general model does not provide a special mechanism to realize the control unit. It is an important feature of our design environment to modify the architecture according to the characteristics of the compiler. We therefore defined a set of instances of the CRC model that feature a configurable finite state machine (FSM) to implement the control unit. This set of instances is realized by a synthesizable Verilog model that consists of an array of uniform PEs (see Fig. 3) that are connected by a nearest neighbor interconnect network. The memory is omitted in these first instances and we decided to integrate a separate FSM within each PE. We experienced that an FSM that is shared by multiple PEs is actually more convenient for application mapping but the absence of features that are shared among multiple PEs makes our first estimations more scalable. Another alternative that we consider in the CRC model to implement the control unit is a complete microcontroller. In that case the PE array acts as an extremely powerful instruction of the microcontroller potentially executing over multiple clock cycles of the controller.

A mechanism to configure the context memory and the FSM from outside at boot time of the device, the separation of data and 1-bit status signals, and multiplexers to select the data and status input signals for the FU are further features of the PE that are not defined in the general CRC model but are implemented in the presented set of instances. The FU supports all operators of the C language except for / and %. All operands are assumed to be of an unsigned integer type of the size defined by the data path width. The data path width is a parameter of the Verilog model. In Fig. 3, the variable width of the data path is marked by the parameter  $D$ . The number of data and status registers ( $\#regs$  in Fig. 3) and the number of states and contexts are additional parameters the Verilog model must be configured with in order to obtain a specific instance of the CRC model. Generally speaking, the FSM of a PE can be implemented to select a context as a function of the state or as a function of the state and the input, implementing Moore and Mealy machines respectively. For a Moore machine it is not possible to provide more contexts than states and for a Mealy machine there is no constraining relation between the number of states and the number of contexts. For the set of instances presented in this paper, we have implemented the FSM as a special case of a Moore machine where the context is directly determined by the state, i.e., a Medvedev machine. Consequently, the number of states is equal to the number of contexts and the time needed for

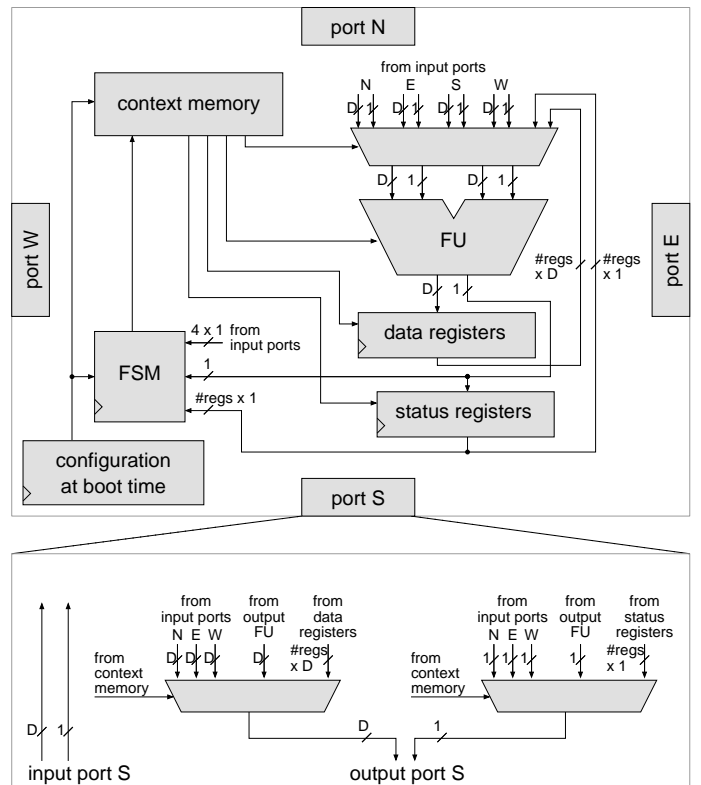


Fig. 3. A PE of the first instances of the CRC model.

reconfiguration can be reduced compared to the other FSM types.

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 status outputs of the FU if the result is to be stored in a register of the PE. The 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 lower part of Fig. 3 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. When operator chaining is performed we take into account the delay of the PEs executing operations in the FU and the delay of PEs that provide nearest neighbor connections to

calculate the delay of the longest combinational path. We therefore use the general term *chain* in the remainder of this paper to denote a chain of PEs that realize an operator chain, no matter if a PE performs an operation or only provides a connection between two other PEs. A PE is able to simultaneously execute an operation in the FU and to route data and status signals from one neighbor to another.

## V. RESULTS

To perform the architecture specific tasks in the design environment it is mandatory to have detailed cost functions for the target architecture. It is not our goal to express the cost functions in terms of mathematical functions. We rather synthesize instances of the CRC model on demand and keep the results in a database as described in section III.

In the following, we present results from area, speed, and power estimations that we obtained by synthesizing PEs from the set of instances described in the previous section. The number of data and status registers is set to twelve and the number of states and contexts is set to 16 for all results presented in this paper. The synthesis was done using a commercial logic synthesis tool and a 0.13 micron standard cell technology library. We used a top-down synthesis strategy for one PE and analyzed the properties of the design after synthesizing it to the gate level.

### A. Cost functions for architecture optimization

When the architecture is modified during the refinement process it is necessary to identify parts of the architecture that provide a high potential for optimization. After modifications are made it is necessary to be able to assess the results of the modification. Eventually, this assessment must include running applications that are mapped by the compiler in order to cover all aspects of the resulting system. The execution time of an application, for instance, can not only be rated by the maximum clock speed at which the architecture can be run. Since not all parts of the compiler have been implemented yet, it is currently not feasible to frequently map a large number of applications as eventually required in our design environment. In this early phase of the project the basic properties of the architecture are analyzed and visualized by diagrams and tables that are automatically generated after synthesis.

**Area:** Fig. 4 depicts the area composition of a PE for a data path width of 8, 16, and 32 bits enforcing a minimum area constraint. The area needed to configure the PE at boot time of the device is mostly consumed by a shift register that stores the address and content of one line in the context memory. Its size is independent of the data path width and it could be reduced by deploying a more sophisticated configuration protocol. The area of the FSM and the context memory is also independent of the data path width. The portion of the FSM is small compared to the context memory. About half of the context memory and about 2/3 of the multiplexer area are used to implement the nearest neighbor interconnect network. Especially for

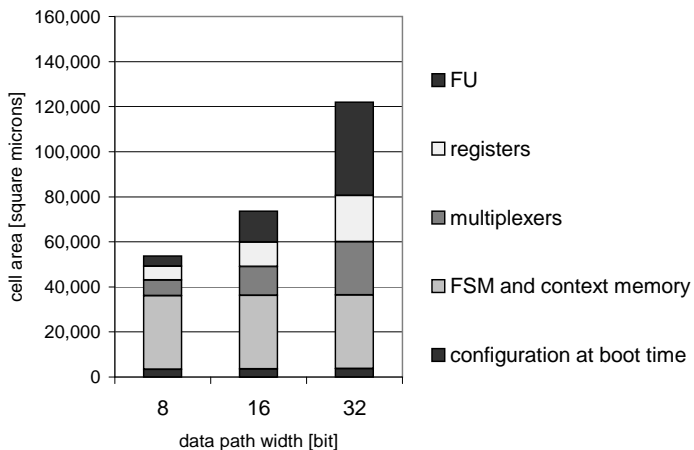


Fig. 4. Area composition of a PE targeting minimum area.

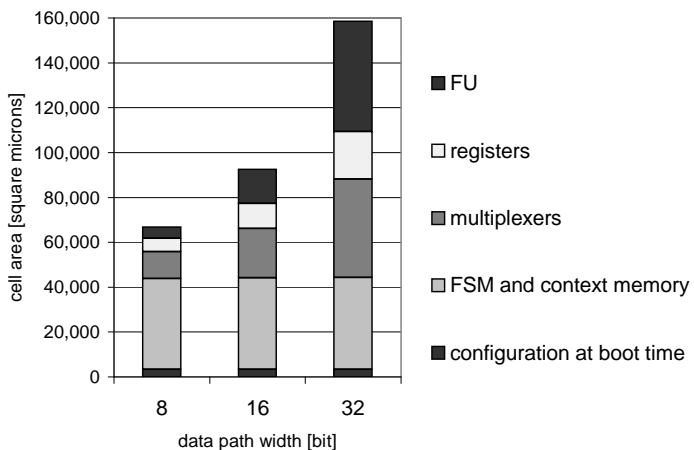


Fig. 5. Area composition of a PE targeting a maximum clock frequency of 200 MHz.

small data path widths the reconfigurable interconnect network constitutes the highest potential to optimize the area. It should be noted that the FSM, the context memory, and the multiplexers are not entirely overhead imposed by the feature of processor-like reconfiguration. In an ASIC or FPGA implementation area is also needed to implement the FSM and to route signals in the data path depending on the state.

In the presented architecture instances we have deployed multipliers that support operands with full data path width. For a 32 bit data path, the FU has a strong impact on the total area with the multiplier consuming by far the most area. It should therefore be well considered if multipliers are actually needed in a domain specific architecture or if their data path width can be reduced.

The same area estimations have been performed targeting a maximum clock frequency of 200 MHz. The results are depicted in Fig. 5. Compared to the instance with minimum area, especially the multiplexers and the FSM and context memory have increased area. This emphasizes the statement that the reconfigurable interconnect network

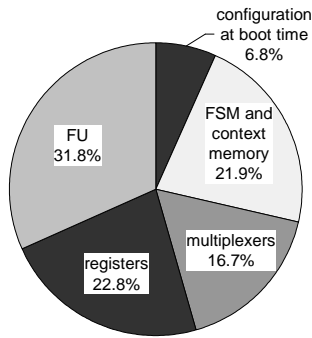


Fig. 6. Breakdown of power dissipation of a PE running at 200 MHz.

constitutes the highest potential to optimize the area.

**Power:** For the power estimations the data path is set to a width of 32 bits targeting a maximum clock frequency of 200 MHz. The figures for power dissipation are obtained from the static estimations of the logic synthesis tool. Power dissipation highly depends on the switching activity of a circuit. Based on the synthesis tool’s default values for switching activity the PE dissipates 12.25 mW when running at a clock frequency of 200 MHz. Fig. 6 shows the power dissipation of the different modules of a PE in percent of the total power dissipation. When the same PE is running at a clock frequency of 100 MHz and 50 MHz a power dissipation of 6.16 mW and 3.12 mW respectively is estimated. In these cases the power dissipation is a little more than half the power dissipation at the the double clock speed. This is due to the static leakage power of 0.071 mW which is independent of the clock frequency. The percent values depicted in Fig. 6 are only slightly different for the lower clock frequencies.

The mechanism to configure the context memory and the FSM from outside at boot time is estimated to dissipate 6.8 percent of the total power. For smaller data path widths the configuration at boot time is estimated to dissipate up to almost one third of the total power. But this part of the circuit is actually not needed during normal operation. We therefore work on instances of the CRC model where switching activity for that part is eliminated after booting the device in order to reduce power dissipation. As in the case of area, the multiplexers together with the context memory, i.e. the reconfigurable interconnect network, provide a high potential to reduce power dissipation. Another step that we consider is to eliminate switching activity for functions of the FU that are not used in a clock cycle by gating the inputs of submodules of the FU.

We are aware that the static power estimation is only a rough estimation. We currently work on incorporating the switching activity that actually occurs when an application is running on the architecture. This more accurate modeling is not only necessary to optimize the power dissipation of the architecture. It is also an important prerequisite to explore power saving techniques in the compiler.

**Speed:** The arithmetic and logic functions of the FU by itself provide full ASIC speed. Compared to a Xilinx

Virtex-II FPGA that is manufactured in a 0.15 micron process technology using 0.12 micron transistors we have measured a speed up of about 10x. For example, when targeting minimum area a 32-bit carry-look-ahead adder has a delay of 35.78 ns on the FPGA and a delay of 3.14 ns when the 0.13 micron standard cell technology is used. But a PE of the CRC model causes additional delay due to the multiplexers used for routing the signal and the time needed for reconfiguration during run-time. The additional delay of a PE has a relatively low impact for arithmetic operations and there is still a significant speed up compared to FPGAs. For simple logic functions this additional delay dominates the overall delay of a PE. We therefore consider the presented architecture instances in particular suitable for applications that feature a high rate of arithmetic operations. Detailed timing information for a PE is provided below.

Similar to what we have pointed out in our area evaluation, the components involved in processor-like reconfiguration might also be present in a circuit that is implemented by an ASIC and in particular in a statically reconfigurable architecture.

### B. Cost functions for the compiler

The techniques from C-based synthesis that we use in our compiler require detailed timing information to calculate the delay of the data paths of an application. Table I shows the delays for a PE performing operations in the FU. Table II lists the delays if the PE is providing nearest neighbor connectivity (*nn operations* in the following). For the delay estimations, the same PE as for the power estimations was used, i.e data path width set to 32 bits and 200 MHz target clock speed. For each operation the delays are itemized for different paths in the PE. For the presented results only the maximum clock frequency at which the architecture can run was optimized. As a consequence, mostly the multiply operation was optimized for speed by the synthesis tool. To improve the results with respect to operator chaining more sophisticated timing constraints can be applied to the design.

The multiply operation is split into two operations *\*l* and *\*h*, where *\*l* calculates the low order bits of the product and *\*h* the high order bits. This way, a multiplication of full data width operands can be performed. If the full result of the product is needed it can be executed by two different PEs in parallel. Addition and subtraction can be used with carry output (*+co*, *-co*) and without (*+*, *-*). For the instance analyzed here, the delays are the same for these operations with or without using the carry output. Other instance that we analyzed do have different delays for the two cases. The *shift* operation is used for left and right shift by interpreting the shift amount as a signed operand. This can be resolved by the compiler in order to support the *<<* and *>>* operators. The relational operators are implemented directly by the FU producing a status signal depending on the result of the comparison. The *AND*, *OR*, *XOR*, and *NOT* operations are implemented for the data and the status inputs of the FU. They can be used

TABLE I  
TIMING RESULTS FOR A PE PERFORMING OPERATIONS IN THE FU.

	state-ext [ns]	state-reg [ns]	ext-ext [ns]	ext-reg [ns]	ext-state [ns]	state-state [ns]
<i>*l</i>	4.95	4.99	3.59	3.61	-	-
<i>*h</i>	5.01	4.97	3.62	3.61	-	-
<i>+co</i>	4.03	3.99	2.49	2.45	2.70	4.21
<i>-co</i>	4.07	4.04	2.65	2.61	2.91	4.34
<i>+</i>	4.03	3.99	2.49	2.45	-	-
<i>-</i>	4.07	4.04	2.65	2.61	-	-
<i>shift</i>	4.43	4.46	2.92	2.96	-	-
<i>==</i>	4.25	4.09	2.74	2.60	3.04	4.55
<i>!=</i>	4.20	4.08	2.71	2.57	3.02	4.50
<i>&gt;</i>	4.24	4.09	2.79	2.64	3.10	4.55
<i>&gt;=</i>	4.47	4.32	2.96	2.81	3.27	4.78
<i>&lt;</i>	4.36	4.24	2.88	2.73	3.17	4.67
<i>&lt;=</i>	4.37	4.21	2.92	2.77	3.23	4.67
<i>AND<sub>d</sub></i>	3.33	2.93	1.36	1.43	-	-
<i>OR<sub>d</sub></i>	3.33	2.93	1.36	1.43	-	-
<i>XOR<sub>d</sub></i>	3.33	2.97	1.43	1.48	-	-
<i>NOT<sub>d</sub></i>	3.33	2.63	1.14	1.21	-	-
<i>AND<sub>s</sub></i>	2.85	2.69	1.31	1.16	1.62	3.15
<i>OR<sub>s</sub></i>	2.92	2.76	1.43	1.28	1.73	3.22
<i>XOR<sub>s</sub></i>	2.95	2.83	1.53	1.37	1.83	3.23
<i>NOT<sub>s</sub></i>	2.87	2.75	1.42	1.26	1.72	3.12
<i>sel</i>	3.46	3.46	1.97	1.97	-	-
<i>in1<sub>d</sub></i>	3.33	2.55	1.22	1.17	-	-
<i>in1<sub>s</sub></i>	2.92	2.76	1.43	1.28	1.73	3.22

to map the corresponding bitwise and logical operators of the C language. For these operations as well as for the nn operations, the indices *d* and *s* are used to denote operations for data and status signals respectively. In addition to the C operators, the FU provides a select function (*sel*) to choose between one of the two inputs depending on the status input. The select function is in particular useful to support speculative parallel execution of different branches in the control flow of an application. Since only the output of the FU can write to a register the FU also allows it to pass one input directly to the output (*in1<sub>d</sub>* and *in1<sub>s</sub>*). The different paths in the PE are described below.

*state-ext* is the delay from the rising clock edge until the result of the operation is available at the output ports of the PE. This delay is composed of the time for a context switch, the time to perform the operation, and the routing to one or more of the output ports of the PE. Our compiler assumes currently that there is a context switch every clock cycle even if the state stays the same in consecutive clock cycles. Therefore, it uses the state-ext time to determine the delay of the first element in a chain. For the first element in a chain the possible inputs are registers inside the

TABLE II  
TIMING RESULTS FOR A PE PERFORMING NEAREST NEIGHBOR CONNECTIONS

	state-ext [ns]	state-reg [ns]	ext-ext [ns]	ext-reg [ns]	ext-state [ns]	state-state [ns]
<i>nnN<sub>d</sub></i>	3.21	-	0.32	-	-	-
<i>nnE<sub>d</sub></i>	3.21	-	0.32	-	-	-
<i>nnS<sub>d</sub></i>	3.21	-	0.27	-	-	-
<i>nnW<sub>d</sub></i>	3.21	-	0.27	-	-	-
<i>nnN<sub>s</sub></i>	2.35	-	0.34	-	0.58	2.32
<i>nnE<sub>s</sub></i>	2.35	-	0.34	-	0.65	2.32
<i>nnS<sub>s</sub></i>	2.35	-	0.34	-	0.65	2.32
<i>nnW<sub>s</sub></i>	2.35	-	0.32	-	0.63	2.32

PE and device ports. The delay from the registers to the multiplexer in front of the FU is by far lower than the time for the context switch since all register outputs are directly wired to the multiplexer. For device inputs we assume that the data is available at the multiplexer within the time needed for a context switch. The state-ext delay therefore covers both cases. For the nn operations, state-ext is the time to propagate the content of an internal register or an input port of the device to the specified PE output port (N, E, S, and W) without using the FU.

Likewise, *state-reg* is the time from the rising clock edge until the result of the operation is available to be stored in one of the registers of the PE. This delay includes the setup time of the register. In this case no chaining is performed for the operation of the PE. For the nn operations this delay is not defined since registers can only be written by the output of the FU.

*ext-ext* is the delay of the operation if source and destination are external to the PE. This delay is used by the compiler if the operation is in the middle of a chain. As can be seen by the varying differences between state-ext and ext-ext it is sensible to analyze state-ext explicitly for each operation instead of adding a fix delay for a context switch to ext-ext. For the nn operations, this is the delay if a PE is used to connect two other neighboring PEs. In this case N, E, S, and W specify for which input port of the PE the delay is given. The delay of the nn operations depends also on the output port of the PE. But there is only little difference between the delays for different destinations and the table shows only the maximum of the three possible values.

*ext-reg* is the delay if the input data is the output of a neighbor PE and the result of the operation is written to a register inside the PE. This means that the PE is at the end of a chain. ext-reg is not defined for the nn operations for the same reason as for state-reg.

*ext-state* is the delay if the operation takes its operands from a neighbor PE and the result is used to determine the next state, i.e., until the new state is available to be stored in the register that holds the current state. Since the next state is determined by the status signals, this delay is only

defined for operations that have a status output. For the nn operations this is the delay if a neighbor PE provides the status information to determine the next state of the PE.

*state-state* is composed of the time for a context switch, the time to perform the operation, and the time to determine the next state depending on the result of the operation. As in the case of state-reg no chaining is performed for the operation of this PE. In contrast to state-reg, the state-state delay is used by the compiler if the status information to determine the next state is also taken from within the PE. This difference is illustrated by the two examples below. The delay for the nn operations is actually not used by the compiler since it can not contribute to the critical path. It is given in the table because it shows the time for reconfiguration isolated from any operation.

The following examples demonstrate how the compiler uses the delay information to calculate the maximum clock speed for an application.

Example 1: A PE calculates the sum of two operands stored in internal data registers. The result is also stored in an internal register and the carry output is used to determine the next state. The delay determining the maximum clock speed at which this most simple design could run is the maximum of the two timing paths state-reg and state-state of  $+co$ , i.e. 4.21 ns.

Example 2: One PE (PE 1) calculates the sum of two operands stored in internal data registers without using the carry output. If the result is stored in a register within PE 1, state-reg of  $+$  (3.99 ns) is the first timing path that must be considered. A neighbor PE (PE 2) north of PE 1 performs a logical NOT operation using a device port as input. The result of PE 2 is used to determine the next state of PE 1 and PE 2. This yields two more timing paths. One is the state-state delay of  $NOT_s$  (3.12 ns). The other is the aggregate delay of state-ext of  $NOT_s$  (2.87 ns) and ext-state of  $nnN_s$  (0.58 ns), i.e. 3.45 ns. The maximum of the three timing paths is 3.99 ns and therefore the design could run at 250.6 MHz.

## VI. CONCLUSIONS AND FURTHER WORK

One of the first steps in our design flow was the implementation of a synthesizable architecture model that fits the basic features of C-based application mapping. In contrast to other projects that are engaged in the design of reconfigurable architectures we optimize the architecture according to the capabilities of a high-level compiler and the requirements of the applications right from the start. With the area, speed, and power estimations from a commercial logic synthesis tool we can carry on with the development of the compiler that needs realistic values for the characteristics of the architecture. The speed estimations demonstrate that the overhead imposed by processor-like reconfiguration is reasonable and that for applications that feature a high rate of arithmetic operations a significant speed up compared to FPGAs can be achieved. This kind of applications can be found in many applications domains that we target. In order to also take into account the wire

delays that are an important factor for deep sub micron technologies we will extend our current synthesis flow to the steps of place and route. Further work also includes enhancements of the architecture like integration of memory blocks and more sophisticated interconnect networks.

## 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] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Design, Automation and Test in Europe*, 2001.
- [2] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. A framework for object oriented hardware specification, verification, and synthesis. In *Design Automation Conference*, 2001.
- [3] M. Motomura. A dynamically reconfigurable processor architecture. In *Microprocessor Forum*, 2002.
- [4] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, “Cyber”. In *Design, Automation and Test in Europe*, 1999.
- [5] V. Baumgarte, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2001.
- [6] IPFlex, Inc. <http://www.ipflex.com>.
- [7] picoChip Designs Limited. <http://www.picochip.com>.
- [8] QuickSilver Technology, Inc. <http://www.qstech.com>.
- [9] A. DeHon. Reconfigurable architectures for general-purpose computing. Technical Report AITR-1586, MIT Artificial Intelligence Laboratory, September 2 1996.
- [10] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, F. Kurdahi, E. Filho, and V. Castro-Alves. The morphosys dynamically reconfigurable system-on-chip. In *The First NASA/DoD Workshop on Evolvable Hardware*, 1999.
- [11] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.
- [12] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac—configurable custom computing. In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.
- [13] A. Abnous and J. Rabaey. Ultra-low-power domain-specific multimedia processors. In *IEEE VLSI Signal Processing Workshop*, 1996.
- [14] A. Stammermann, L. Kruse, W. Nebel, A. Pratsch, E. Schmidt, M. Schulte, and A. Schulz. System level optimization and design space exploration for low power. In *International Symposium on System Synthesis*, 2001.
- [15] O. Bringmann and W. Rosenstiel. Cross-level hierarchical high-level synthesis. In *Design, Automation, and Test in Europe*, 1998.
- [16] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel. A design environment for processor-like reconfigurable hardware. In *IEEE International Conference on Parallel Computing in Electrical Engineering*, 2004.