Optimization of Area and Performance by Processor-Like Reconfiguration

Tobias Oppold, Sven Eisenhardt, Wolfgang Rosenstiel

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

Abstract

It is well known that the area efficiency of a digital circuit can be improved by reconfiguration due to the reuse of resources. In this paper, we show that this benefit can be achieved for a wide range of applications if the reconfiguration can take place within each clock cycle, and we quantify the benefit by area estimations from a synthesizable architecture model. Although reconfiguration typically involves a decrease of performance, we show how performance can actually be increased by redirecting communication through the time domain. This increase is quantified by estimations from a silicon-proven commercial architecture and its associated compiler.

1. Introduction

Reconfigurable systems provide the ability to reuse architectural resources over time. This reuse is enabled by a variety of architectures using different models of reconfiguration [4, 15]. For statically reconfigurable architectures (e.g. Xilinx XC4000 FPGAs), execution must be interrupted for the reconfiguration process and reuse typically happens in the range of minutes or even much longer.

Dynamically reconfigurable architectures (DRAs) can be reconfigured during run-time to enable a higher degree of reuse. DRAs include partially reconfigurable (e.g. Xilinx Virtex FPGAs), pipeline reconfigurable (e.g. PipeRench [3]), and multi-context architectures (e.g. DPGA [5]).

More recently, a growing number of commercial coarse grained architectures with more or less different models of reconfiguration appeared on the market (NEC-DRP, PACT-XPP, IPFlex-DAP/DNA, MathStar-FPOA, to name just a few). Such architectures are often available as parameterizable IP-cores for System-on-Chip design rather than as a off-the-shelf product. Some of these architectures can be reconfigured within one clock cycle. We call such architectures *processor-like reconfigurable* architectures.

For lack of standardized benchmarks, it is hard to say which of the existing reconfigurable architectures provides the best performance/cost ratio for a large set of applications. This paper is focused on the costs and benefits of processor-like reconfiguration in terms of silicon area and performance compared to other models of reconfiguration. To emphasize the aspect of reconfiguration, we try to mask out other features like the width of the data path or the operations supported by the functional units.

We show that by processor-like reconfiguration the area requirements of a coarse-grained reconfigurable IP-core can be optimized considerably while meeting a given performance constraint. For this purpose we use our modifiable and synthesizable architecture model (Configurable Reconfigurable Core, CRC) that allows it to generate architecture instances with the number of contexts and processing elements (PEs) being customized for a single application under a given performance constraint. The cell area of such a core is estimated at the gate-level and compared to a reconfigurable core with only one context and the number of PEs also being customized for the application.

The modifiable CRC model enables a detailed evaluation of area which we can not accomplish with commercial architectures since detailed area information is usually not available. On the other hand, using the stable tool chain of commercial architectures enables the evaluation of complex applications more efficiently. Since NEC's DRP (Dynamically Reconfigurable Processor) architecture [9] is very similar to the CRC model with respect to run-time reconfiguration [10], we can use the silicon-proven DRP and its associated tools [13] to get additional evaluation data.

The DRP compiler in particular provides timing information including the effects of placement and routing. We therefore use the DRP system to demonstrate that the time needed for processor-like reconfiguration can be compen-

1-4244-0910-1/07/$20.00

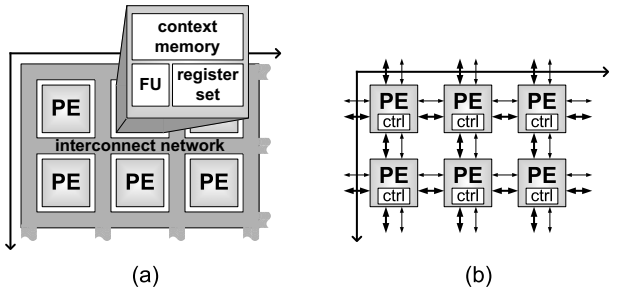

Figure 1. General CRC model (a) and overview of the CRC-A template (b).

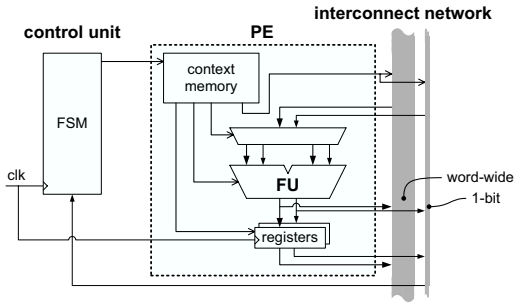

Figure 2. Features common to the CRC-A template and the DRP architecture.

sated by redirecting communication through the time domain and that a processor-like reconfigurable architecture can even outperform architectures that are not reconfigured during run-time.

The CRC model and the DRP architecture are described in the next section. In Section 3, we present our evaluation approach including techniques for mapping applications. Related work is also discussed in that section. Area and performance evaluations are presented in Section 4 and 5, respectively.

2. Architecture Model

The CRC model was developed to represent a wide range of processor-like reconfigurable architectures. In its most general specification, only a few features are defined. As depicted in Fig. 1, it consists of a rectangular array of processing elements (PE) that are connected by a reconfigurable interconnect network. Each PE consists of a functional unit (FU) for word-wide arithmetic and logic operations, a register set, and a context memory that defines several configurations for the PE. A context is selected by a control unit which can vary significantly for the various architectures. So does the interconnect network and therefore both are not further specified in the general CRC model.

Based on that general specification we have mapped the operations and control structures of several applications described in C onto the CRC model. The model was successively augmented with features that enable the execution as proposed by the application mapping. These features are implemented as a parameterizable architecture template. We denote this template as CRC-A since it is the first out of a set of architecture templates that we implemented in Verilog.

Fig. 2 depicts the features that are common to the DRP architecture and the CRC-A template. The interconnect network is subdivided into word-wide data channels and 1-bit status signals but not further detailed. For the FU and the registers, the word-wide data flow is also separated from the 1-bit signals. The output of the FU can be stored in a register and it can be fed into the interconnect network to execute two or more operations in a chain of FUs. For the control unit, it is only specified that it implements a finite state machine (FSM) controlled by the 1-bit status signals and that an entry of the context memory is selected by the FSM at the beginning of each clock cycle.

2.1. CRC-A Architecture Template

Fig. 1(b) shows the topology of the interconnect network and the control unit for the CRC-A template. Each PE features its own control unit and the interconnect network implements a nearest-neighbor (NN) network. The PEs at the borders of the array use the NN-network for I/O.

For the CRC-A template we have implemented a number of RT-level components (FUs with and without multiplier, NN-networks with one or two data channels, etc.) that can be combined in various ways so that different architecture instances can be synthesized and analyzed in a highly automated flow with reasonable effort. To create architecture instances, the template is configured by selecting RT-level components and setting parameters of the Verilog code (e.g., the word length and the number of contexts).

2.2. NEC-DRP

For the DRP architecture, there is one control unit ("state transition controller") for an array of 8x8 PEs as depicted in Fig. 3. The array is surrounded by embedded memory blocks.

The interconnect network implements a more sophisticated topology compared to the simple NN-network of the CRC-A template. In reference to Fig. 2, the word length of the DRP is 8 bits and there are 16 contexts. A DRP "core" is composed of one or more such DRP "tiles". We use the

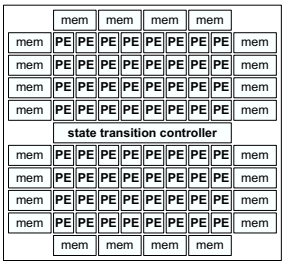

Figure 3. A tile of the DRP architecure [9].

DRP-1 prototype core that consists of 8 tiles, i.e. 512 PEs in total, and is manufactured in a 150 nm CMOS technology.

The compiler for the DRP partitions an application written in C into several configuration contexts automatically. NEC developed this compiler based on their hardware synthesis tool Cyber [16].

3. Evaluation Approach

As outlined in the introduction, our evaluation of area requirements is based on the analysis of reconfigurable cores customized for a single application associated with a performance constraint. The design of such a core highly depends on how the application is eventually mapped to it. Mapping techniques to take advantage of processor-like reconfiguration are therefore crucial for our evaluation approach and described in detail later in this section.

We consider applications written in C and focus especially on techniques for scheduling operations for execution in different contexts. These techniques minimize the number of required FUs by reusing them in different clock cycles which also leads to reuse of the interconnect network. Based on the application mapping, we create a customized instance of the CRC model that features only the required number of FUs and contexts.

By applying only techniques that do not take advantage of processor-like reconfiguration, the same application can be mapped to a reconfigurable core with only one context and the number of FUs also being customized for the application. For comparison with a customized processor-like reconfigurable architecture, we implement such a core as a coarse-grained architecture with the lowest overhead for reconfiguration in terms of area, i.e., as a statically reconfigurable variant of the CRC model. For such an architecture, resources could be reused by multiplexing the data path using PEs that provide multiplexer functionality. This is commonly done by high-level synthesis tools targeting fine-grained FPGAs which can implement multiplexers efficiently. Like other coarse-grained architectures, the CRC-A template provides multiplexing as one of the mutually exclusive operations of the FU and therefore supports this technique not efficiently. Since it often results in even higher resources requirements it is not considered in our approach.

For the evaluation of performance, the interconnect delay must not be neglected since it is dominant for the architectures that we consider [13]. Our tools for placement and routing are currently in a rather experimental state [2]. Since these steps influence the performance results significantly, we use NEC's DRP and its stable tool chain for the performance evaluation. However, the DRP compiler does not provide all scheduling techniques that we apply. We therefore provide the scheduling of the operations in the C code for the DRP compiler by breaking up data dependencies and rearranging the original source code where necessary. By this approach, we can bypass the scheduler of the DRP compiler and use the compiler for technology mapping, placement, and routing.

To specify a performance constraint for the evaluation, we use the initiation interval (*II*), i.e. the number of clock cycles that are available to consume a set of input values. For the area evaluation, the clock frequency is not specified and we rather compare IP-cores based on clock cycles. For the performance evaluation, we try to achieve the maximum clock frequency for a given *II*.

3.1. Techniques for Data Flow

The most interesting aspect of processor-like reconfigurable architectures is certainly *multi-context execution* since it provides an additional degree of freedom for application mapping compared to statically reconfigurable architectures. Early work on fast reconfiguration has been presented by DeHon [5] and Trimberger et al. [14]. But the principle of multi-context execution is actually well known for much longer from von Neumann architectures where the context is changed with each instruction. To realize multi-context execution, we schedule all instructions that can be executed in parallel for execution in one control step. Each of these control steps is assigned to a separate context and the contexts are executed sequentially.

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

If the *II* allows distributing the operations over multiple clock cycles, we combine the previous techniques as *multi-context pipelining*. This corresponds to the software pipelining techniques used by software compilers. Modulo scheduling algorithms are used by software compilers for pipelining loops at various *II*s. Mei et al. present a modulo

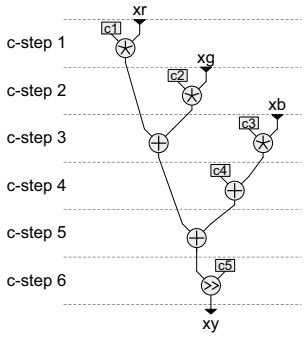

Figure 4. Scheduling of operations for II=3.

scheduling algorithm for mapping loop kernels to coarse-grained reconfigurable architectures in [8].

To illustrate multi-context pipelining, Fig. 4 shows the result of an ASAP (as soon as possible) scheduling for *II*=3 for the following example:

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

$c1$ to $c5$ are constants and xr, xg, and xb are the inputs that are read in three different control steps (c-steps) due to the *II* constraint. The sequential execution of all six c-steps would result in *II*=6. Therefore, c-steps 1 & 4 are executed in parallel in context 1, and c-steps 2 & 5 and 3 & 6 are executed in parallel in context 2 and context 3, respectively, to achieve *II*=3.

3.2. Techniques for Control Flow

Branches in the control flow can be resolved by *spatially multiplexing* the data path. This transforms the control flow graph of an application into a pure data flow graph that can be further processed as described above. This approach is commonly used for reconfigurable architectures, e.g. by Huang and Malik [7], and is also well known from hardware design.

Alternatively, the branches can be assigned to different contexts, i.e. temporally multiplexed, to minimize the number of required FUs without impact on the performance since always only one of the branches is actually needed during execution. We call this technique *multi-context control flow branches* which corresponds to the conditional jump instructions of von Neumann architectures. In [12], Rivera et al. propose a configuration scheduler for conditional branch execution which assumes that the computation is interrupted for switching configurations. For processor-like reconfigurable architectures like the CRC model and NEC's DRP architecture, reconfiguration is part of the regular execution. Branches in the control flow can thus be handled at a finer granularity.

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

4. Evaluation of Area

To quantify the area trade-off between statically and processor-like reconfigurable architectures, we used instances of the CRC-A template based on four different PEs. To estimate the area of statically reconfigurable architectures, the control unit and the context memory have been removed from the PEs of the CRC-A template and each PE features one data and one status register. The configuration data to control the operation performed by the FU as well as the interconnections is directly taken from a shift register inside the PE which is otherwise used to transfer the configuration data from outside the core to the control unit and the context memory of the PE. For the processor-like reconfigurable architectures, PEs of the CRC-A template with 2, 4, and 8 contexts and the same number of states in the control unit as well as data and status registers were used. For all instances the word length is set to 32 bit, each FU features a 16x16-bit multiplier, and the NN-interconnect provides one data channel and one channel for status signals.

We synthesized the PEs using a commercial synthesis tool targeting a 130 nm standard cell technology. For all PEs a target clock speed of 200 MHz was specified. The area of a PE is estimated as the cell area of the resulting gate-level netlist. To obtain area estimations for an architecture instance being composed of several identical PEs, we summed up the area of all PEs being part of the instance.

Table 1 shows the results of applying the techniques described in the previous section to four example applications. For each application, different techniques are used resulting in different requirements in terms of needed FUs and contexts. For each application/technique pair, the area of a customized IP-core featuring exactly the required number of FUs is provided in the table. The number of contexts provided by the core, and with it the number of states and registers, is rounded up to the next power of two. The interconnect network and register requirements are not considered and we set the number of PEs equal the number of required FUs. Since the interconnect resources can be reused in different contexts, similar penalties resulting from poor

application	techniques	*II*	FUs	contexts	area of IP-core with min. # FUs	area compared to static arch.
rgb2yiq	pipelining	1	21	1	0.610 mm^2	100.0 %
rgb2yiq	multi-context pipelining	3	7	3	0.322 mm^2	52.7 %
ellipf	pipelining	1	26	1	0.756 mm^2	100.0 %
ellipf	multi-context pipelining	2	13	2	0.468 mm^2	61.9 %
ellipf	multi-context pipelining	4	7	4	0.322 mm^2	42.6 %
ellipf	multi-context pipelining	8	4	8	0.280 mm^2	37.0 %
rgb2cmyk	spatially multiplexed data path / pipelining	1	9	1	0.262 mm^2	100.0 %
rgb2cmyk	pipelined multi-context control flow branches	1	7	3	0.322 mm^2	132.0 %
resampling	spatially multiplexed data path / pipelining	1	39	1	1.134 mm^2	100.0 %
resampling	pipelined multi-context control flow branches	1	28	2	1.007 mm^2	88.9 %

Table 1. Area results for example applications.

routability can be expected for the different architectures customized for one application. The implications of excessive register requirements are discussed in [10]. The last column in the table compares this IP-core to a customized statically reconfigurable architecture that uses the same *II* but does not reuse the FUs.

4.1. Data Flow Examples

The first example in table 1 (rgb2yiq) is the loop kernel of the RGB to YIQ conversion which is part of the EEMBC Consumer Benchmark [6]. The loop kernel contains 21 operations without branches in the control flow so that 21 FUs are required for fully pipelined execution (*II*=1). Since processor-like reconfiguration can not be applied in this case, the customized IP-core features a single context and no control unit and thus requires 100 % of the area required for the statically reconfigurable alternative. If the three input values are read in three different clock cycles (*II*=3), the operations can be distributed over these three clock cycles using multi-context pieplining. This requires seven FUs and three contexts resulting in an IP-core that features seven PEs with four contexts. The IP-core for *II*=3 requires about half the area of a statically reconfigurable architecture since the latter does not benefit from the increased *II* and therefore has the same area as for *II*=1.

The second example (ellipf) is the fifth order elliptical wave filter from the high-level synthesis benchmark suit [1]. It contains 26 operations without branches in the control flow. The results in the table are similar to the rgb2yiq example. To quantify the area penalty resulting from contexts being implemented in an IP-core but not used by the application, a more detailed evaluation of the ellipf example is depicted in Fig. 5. For this additional evaluation, IP-cores with 2, 4, and 8 contexts as well as a static alternative were targeted for all four *II*s (1, 2, 4, and 8). In contrast to the approach described in Section 3, the minimal number of FUs

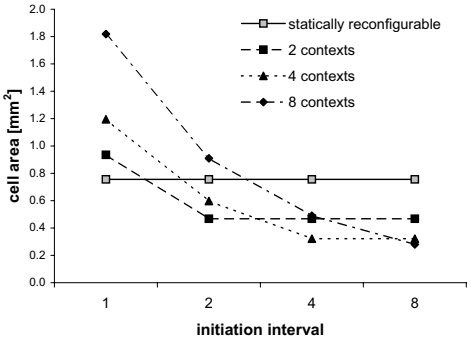

Figure 5. Detailed evaluation of the ellipf example.

was determined based on the *II* and the context constraint for each *II*/architecture pair.

For *II*=1, all architectures require 26 FUs so that the statically reconfigurable architecture yields the lowest area and the IP-core with 8 contexts requires more than double the area. For *II*=2, the core with 4 contexts already requires less area than the statically reconfigurable architecture although only two contexts are used by the application. The 8-context architecture pays off only for *II*=4 and *II*=8.

4.2. Control Flow Examples

The third example in table 1 (rgb2cmyk) is the loop kernel of the RGB to CMYK conversion which is also part of the EEMBC Consumer Benchmark. It contains three if/else statements but only few operations within the branches:

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

The lowest area for *II*=1 is achieved if the branches in the control flow are spatially multiplexed in the data path and the execution is pipelined afterwards. By mapping the branches to different contexts, two FUs can be saved. One FU can be saved because the two compare operations `c<y` and `m<y` can be assigned to different contexts sharing one FU. The other FU can be eliminated by moving the final calculation of `c`, `m`, and `y` into the branches so that in each branch one subtraction can be replaced by constant assignment to zero. But 7 PEs with 4 contexts require more area than 9 PEs of a statically reconfigurable architecture so that processor-like reconfiguration does not pay off for this example.

The fourth example (resampling) is an implementation of the resampling stage of the ray casting algorithm as described in [11]. In this example, two different filter kernels to perform the resampling are implemented. During execution, only one of the filters is applied so that they can be mapped to two different contexts without increasing the *II*. For this example, only a moderate area improvement can be achieved by processor-like reconfiguration compared to a statically reconfigurable architecture because the operations are not evenly distributed over the two branches (28 vs. 11).

The two control flow examples show that it is highly application-dependent whether mapping control flow branches to multiple contexts leads to an area improvement over a statically reconfigurable architecture. However, it is a useful technique in general if the targeted IP-core features multiple contexts, e.g., because other parts of the application are mapped to different contexts.

5. Evaluation of Performance

For deep sub-micron process technologies, the interconnect delay contributes significantly to the overall delay of digital circuits. In addition to the metal wire delays, the reconfigurable routing switches contribute to the interconnect delay of reconfigurable architectures.

Processor-like reconfiguration allows it to utilize reconfiguration as a third dimension for routing by redirecting communication through the time domain. By doing so, the connection lengths may be reduced as illustrated in Fig. 6 for an 8-point fast Fourier transform (FFT) implementing the Cooley-Tukey algorithm. When moving from *II*=1 to *II*=2, the longest connections can be reduced to nearest-neighbor connections by executing the upper and the lower half of the graph in different contexts.

By doubling the number of contexts again (*II*=4), the connections in the middle of the graphs can also be reduced to nearest-neighbor connections. Due to the regularity of the Cooley-Tukey algorithm, the same considerations hold for n-point FFTs at different *II*s in general (with n and *II* being powers of two).

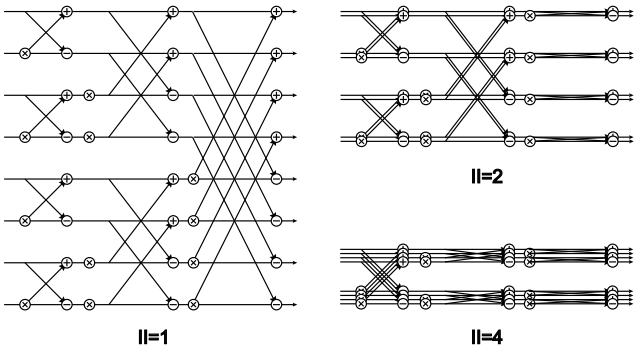

Figure 6. Reduction of an 8-point FFT's interconnect requirements by redirecting communication through the time domain.

For the following experiments, pipelining and multi-context pipelining based on operations as described in Section 3.1 are applied and the resulting clock frequency is estimated. To further increase the clock frequency, additional registers could be inserted between operations to pipeline communication over the interconnect network. This diminishes the impact of interconnect delay in terms of throughput but also increases the latency and requires a larger number of registers which already was a limiting factor for the application mapping.

5.1. Experiments

As explained in Section 3, we used NEC's DRP and its associated tools for the performance estimations. For the experiments we used a simplified version of the FFT. This simplified FFT uses integer arithmetic (add and subtract operations) for the nodes of the graphs in Fig. 6 while the Cooley-Tukey algorithm uses complex arithmetic (multiply, add, and subtract). This is done to focus on the interconnect delay, since otherwise the mapping of the complex operations to the integer FUs of the DRP would require too many PEs to implement larger FFTs.

Fig. 7 shows the clock frequencies achieved for simplified FFTs with 2 to 16 input values (corresponding to 2-point to 16-point FFTs) at different *II*s. The clock frequencies are estimated by the DRP compiler after placement and routing. For the simplified FFT with 16 input values, placement and routing was not possible at *II*=8 for lack of registers and therefore no data point is provided in the figure.

Except for the simplified FFT with 2 input values, for which the interconnect requirements are already very low for *II*=1, a noticeable gain of clock speed can be observed when the *II* is increased from 1 to 2 and to 4. For *II*=8, the

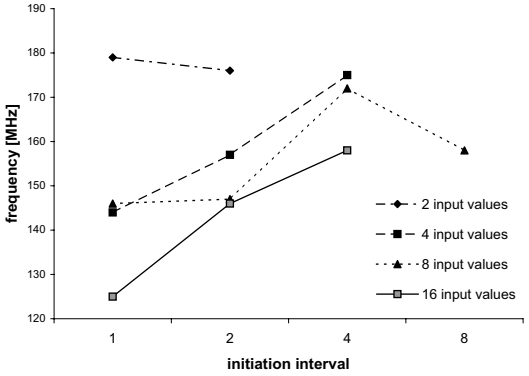

Figure 7. Clock frequencies for the simplified FFT.

clock speed drops below the speed achieved for *II*=4. The irregular characteristics of the curves are discussed in the following.

The pipelined (*II*=1) and multi-context pipelined ($II \geq 2$) execution was achieved by breaking up data dependencies in the C code of the FFT. But we did not force a placement of the operations as proposed by Fig. 6. This allows the DRP compiler to optimize the placement within one context, which in turn leads to a mapping of overlapping operations (Fig. 6, $II \geq 2$) to different FUs possibly far away form each other. Therefore the speed-up is enabled independently of the placement but solely by the reduction of routing complexity. This reduces the benefit of redirecting communication through the time domain compared to the idealized placement suggested by Fig. 6.

In addition to that, the irregular placement of the operations prohibits storing intermediate results locally in the register sets of the PEs that actually perform the operations. Therefore a great number of PEs is used for storing intermediate results only and the placement becomes constrained by the availability of registers rather than by the availability of FUs. This becomes in particular obvious for *II*=8, where placement and routing was not possible anymore (16 input values) or the clock frequency drops below the frequency for *II*=4 (8 input values).

5.2. Comparison to Statically Reconfigurable Architectures

To estimate whether the results for $II \geq 2$ are superior to a statically reconfigurable architecture, it is assumed that a statically reconfigurable architecture similar to the DRP is available. This architecture would be the same as the DRP except for the ability to switch contexts, i.e. the control unit ("state transition controller" in Fig. 3) and the context memory are removed. For this architecture the clock speed can be calculated by equation 1, whereas the clock speed for a processor-like reconfigurable architecture is calculated by equation 2, with t_{rec}, t_{FU}, and t_{ic} being the delay imposed by reconfiguration, calculation, and communication over the innterconnect network, respectively.

$$clk_{static} = 1/(t_{FU} + t_{ic}) \quad (1)$$

$$clk_{proc-like} = 1/(t_{rec} + t_{FU} + t_{ic}) \quad (2)$$

A processor-like reconfigurable architecture outperforms a statically reconfigurable architecture for *II*=n if inequality 3 is true, with $t_{ic,static}$ being the interconnect delay of a statically reconfigurable architecture and $t_{ic,II=n}$ being the interconnect delay of a processor-like reconfigurable architecture for *II*=n.

$$t_{FU} + t_{ic,static} > t_{rec} + t_{FU} + t_{ic,II=n} \quad (3)$$

For the statically reconfigurable architecture, $t_{ic,static}$ can not be decreased by redirecting communication through the time domain, i.e. it is constant for all *II*s of one simplified FFT with a given number of input values. We assume that $t_{ic,static}$ is equal to $t_{ic,II=1}$ as estimated for the DRP so that the break-even point can be expressed by inequality 4.

$$t_{FU} + t_{ic,II=1} > t_{rec} + t_{FU} + t_{ic,II=n} \quad (4)$$

Since we can obtain performance estimations for the DRP only for $t_{rec} + t_{FU} + t_{ic,II=n}$ as a whole, inequality 4 must be rewritten as inequality 5 in order to estimate the performance of a statically reconfigurable architecture.

$$(t_{rec} + t_{FU} + t_{ic,II=1}) - t_{rec} > t_{rec} + t_{FU} + t_{ic,II=n} \quad (5)$$

By transforming inequality 5 to inequality 6, the performance estimations of our experiments can be used to determine the break-even point in terms of time that can be spent for reconfiguration.

$$t_{rec} < (t_{rec} + t_{FU} + t_{ic,II=1}) - (t_{rec} + t_{FU} + t_{ic,II=n}) \quad (6)$$

In [13], the time to select a context of the DRP is reported to be "less than a nanosecond." But the time to select a context is not exactly the same as t_{rec} since the latter is related to an architecture where control unit and context memory are completely absent. Table 2 summarizes the calculated right hand side of inequality 6 based on our experiments. In four of the seven cases t_{rec} can be more than one nanosecond and still the processor-like reconfigurable architecture would outperform an architecture that can be reconfigured only statically.

	II=2	*II*=4	*II*=8
4 input values	0.58 ns	1.23 ns	n/a
8 input values	0.05 ns	1.04 ns	0.52 ns
16 input values	1.15 ns	1.67 ns	n/a

Table 2. Delay penalty that may be imposed by reconfiguration in order to outperform a statically reconfigurable architecture for the simplified FFT.

6. Conclusions and Further Work

The experiments presented in this paper show how to take advantage of processor-like reconfiguration to optimize area and performance. The presented techniques for mapping C code do not rely on application-specific features but are applicable to a wide range of applications if the I/O is constrained by the surrounding system or if/else-statements in the source code split up the control flow.

The comparison of statically and processor-like reconfigurable architectures customized for a single application demonstrate that a significant reduction of area can be achieved by fast reconfiguration while meeting I/O constraints. In ongoing work, we explore architectures that are optimized for an application domain since reconfigurable architectures are usually not intended to be used for just a single application.

Redirecting communication through the time domain can compensate for the performance overhead imposed by processor-like reconfiguration or even outperform a statically reconfigurable architecture. We have deliberately selected FFT as the example application for its regular communication structure. Experiments with other applications actually resulted in slower clock frequencies when the *II* is increased. On the other hand, placement and routing was not targeted on optimizing communication through the time domain and we believe that the potential of this paradigm is not fully exploited yet. We therefore work on compiler techniques which combine scheduling, placement, and routing and take into account the delay estimations from the CRC model.

7. Acknowledgment

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

References

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

[2] J. Brenner, J. van der Veen, S. Fekete, J. Oliveira Filho, and W. Rosenstiel. Simultaneous scheduling, binding and routing for processor-like reconfigurable architectures. In *International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 2006.

[3] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas. Managing pipeline-reconfigurable FPGAs. In *ACM/SIGDA International Symposium on Field- Programmable Gate Arrays (FPGA)*, 1998.

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

[5] A. DeHon. DPGA utilization and application. In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 1996.

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

[7] Z. Huang and S. Malik. Exploiting operation level parallelism through dynamically reconfigurable datapaths. In *Design Automation Conference (DAC)*, 2002.

[8] B. Mei, S. Vernalde, D. Verkest, H. DeMan, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Design, Automation and Test in Europe (DATE)*, 2003.

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

[10] T. Oppold, S. Eisenhardt, and W. Rosenstiel. Design and validation of execution schemes for dynamically reconfigurable architectures. In *International Conference on Field Programmable Technology (FPT)*, Bangkok, Thailand, 2006.

[11] 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)*, Tampere, Finland, 2005.

[12] F. Rivera, M. Sánchez-Élez, M. Fernández, R. Hermida, and N. Bagherzadeh. Configuration scheduling for conditional branch execution onto multi-context reconfigurable architectures. In *International Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, 2006.

[13] T. Toi, N. Nakamura, L. Jing, Y. Kato, T. Awashima, and K. Wakabayashi. High-level synthesis challenges and solutions for a dynamically reconfigurable processor. In *International Conference on Computer-Aided Design (ICCAD)*, Madrid, Spain, 2006.

[14] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997.

[15] 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.

[16] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(12):1507–1522, 2000.