Hyperreconfigurable Architectures as Flexible Control Systems*

Sebastian Lange and Martin Middendorf
Department of Computer Science
University of Leipzig
Augustusplatz 10-11, D-04109 Leipzig, Germany
{langes, middendorf}@informatik.uni-leipzig.de

Abstract: Hyperreconfigurable architectures can change their reconfiguration capabilities dynamically at run-time. For reconfiguration they use two types of reconfiguration steps: i) in hyperreconfiguration steps they change their ability for reconfiguration, ii) in ordinary reconfiguration steps they reconfigure the actual contexts of a computation within the limits that have been set by the preceding hyperreconfiguration step. Hyperreconfigurable architectures have originally been introduced to increase the speed of run-time reconfiguration. In this paper we show that the high flexibility with respect to runtime reconfiguration makes hyperreconfigurable architectures well suited for the control of processes that demand varying amounts of supervision. One advantage of hyperreconfiguration is that the run-time of a control task can be influenced without changing the task itself but only by using different variants of other control tasks that run in parallel. To illustrate the concepts we present the results of simulations with a small hyperreconfigurable architecture where counter and adder control tasks run in parallel.

1 Introduction

Reconfigurable systems offer the advantage of fast hardware execution with the possibility to easily modify the hardware computations. Especially attractive is the high flexibility offered by run-time reconfigurable systems that can dynamically change their internal communication structure and/or their functional units.

Hyperreconfiguration is a new concept for run-time reconfiguration that has been proposed recently to cope with the problem of the increasing amount of reconfiguration data that are necessary to specify the behavior of modern run-time reconfigurable hardware (see [Midd03, LaMi04a]). Other approaches that have been proposed are: i) the use of off-line compression methods applied to the reconfiguration bit stream before it is loaded onto the system [DaPr01, HLR99], ii) to compute the bits which are necessary for reconfiguration directly on chip [KoeTe02, SWMP00, WaDa00]), iii) to clone configurations that

*This work was financed by the German Research Foundation (DFG) through the project "Models and Algorithms for Hyperreconfigurable Architectures" within the priority programme 1148 "Reconfigurable Computing Systems"

are already on the machine [PaBu99], iv) to perform the reconfiguration incrementally [LeWo02].

Since reconfiguration data have to be transferred for every reconfiguration step this large amount of information transfer is especially critical for computations that exploit the full capacity of dynamically reconfigurable architectures by frequent reconfigurations. The basic principle of the hyperreconfiguration approach is to make the ability for reconfiguration itself reconfigurable so that ideally only those parts of the architecture that are needed for reconfiguration are actually available and therefore have to be supplied with reconfiguration data. The reconfiguration potential of an architecture can then be decreased (increased) during such periods of a computation where less (more) reconfiguration features are required. Only the new states of the actually available reconfigurable units have to be defined during a reconfiguration step and therefore the amount of necessary reconfiguration information only depends on the reconfiguration potential that is actually used. For reconfiguration hyperreconfigurable architectures use in addition to the ordinary reconfiguration steps a second type of reconfiguration steps which determine the actual reconfiguration potential of the architecture. These steps are called *hyperreconfiguration steps*.

Hyperreconfigurable architectures have originally been introduced with a focus on increasing the speed of run-time reconfiguration. In this paper we show that the high flexibility with respect to run-time reconfiguration makes hyperreconfigurable architectures well suited to implement controllers for processes with variable demands for supervision. The aim of this paper is not to discuss real life applications but to explain the general concept and to illustrate it with a rudimentary sample architecture.

In the next Section 2 we describe hyperreconfigurable architectures in general. In Section 3 we introduce the concept of using hyperreconfigurable architecture for implementing controllers for processes with variable demands of supervision. Our example architecture is presented in Section 4. Section 5 discusses the results of simulated test runs with the example architecture. The paper ends with a conclusion in Section 6.

2 Hyperreconfigurable Architectures

In this section we give an abstract definition of hyperreconfigurable machines and then describe a specific model for such architectures (for more details and other models see [LaMi04a]). An example architecture that illustrates the definitions is presented in Section 4.

A reconfigurable system is called hyperreconfigurable when its ability for reconfiguration is reconfigurable itself ([Midd03]). Hyperreconfigurable architectures use two types of reconfiguration steps: i) *ordinary reconfiguration steps* allow to change the context of an algorithm during run-time where the context determines the communication structure and/or the functional units, ii) *hyperreconfiguration steps* allow to define the potential for reconfiguration (e.g. the number and size of available reconfigurable units or the properties of the switch boxes for defining connections) that is available for the following ordinary

reconfiguration steps.

In order to guarantee that enough reconfiguration potential is available for each reconfiguration step of an algorithm we assume that an algorithm/computation is characterized by a sequence of context requirements that specifies for every reconfiguration step which reconfigurable features it needs. Formally, let $\mathcal{C}$ be the set of possible context requirements for a hyperreconfigurable machine then an algorithm is characterized by a sequence $C = c_1 \dots c_n$ with $c_i \in \mathcal{C}$, $i \in [1 : n]$. The context requirements are worst case requirements that guaranty a successful computation when satisfied. The actual demand of a computation during runtime might be lower and since it depends on the data it cannot be determined exactly in advance. When the meaning is clear we call the context requirements of an algorithm/computation sometimes simply its contexts. The reason is that each context requirement corresponds to exactly one new context that will be realized during runtime by a reconfiguration operation. This is possible only when the machine is in a state (more exactly in a hypercontext — see definition below) that provides the necessary reconfigurable features, i.e., it satisfies the corresponding context requirement.

A *hypercontext* is characterized by the subset of context requirements in $\mathcal{C}$ that are possible under it. Let $\mathcal{H}$ be the set of possible hypercontexts for a machine. For a hypercontext $h \in \mathcal{H}$ let $h(\mathcal{C}) \subset \mathcal{C}$ be the subset of context requirements that are satisfied by h. The set $h(\mathcal{C})$ is called the *context set* of h. For a sequence $c_1 \dots c_k$ of context requirements and a hypercontext h let $c_1 \dots c_k \subset h(\mathcal{C})$ denote the fact that for each context c_i, $i \in [1 : k]$, $c_i \in h(\mathcal{C})$ holds. For each hypercontext $h \in \mathcal{H}$ there exist the following two costs: i) $init(h)$ are the costs to perform a hyperreconfiguration that brings the machine into hypercontext h, ii) $cost(h)$ are the costs for an ordinary reconfiguration step when the machine is in hypercontext h. The costs might be measured as the number of (hyper)reconfiguration bits that have to be loaded on the machine or as the time for loading them. Thus, during the execution of an algorithm/computation a machine performs operations $h_1 S_1 \dots h_r S_r$ where $h_1, \dots, h_r$ are hyperreconfigurations and S_i stands for a sequence of reconfigurations which use only those parts of the machine that are available within h_i.

Different cost models for hyperreconfigurable machines have been discussed in [LaMi04a]. In this paper we use only the so called *Switch model* which allows to decide for each reconfiguration bit whether it is contained in the hypercontext or not. Thus it is assumed that there exists a set $X = \{x_1, \dots, x_n\}$ of small reconfigurable units which we call switches. The set of context requirements $\mathcal{C}$ and the set of hypercontexts $\mathcal{H}$ are the set of all subsets of X, i.e. $\mathcal{C} = \mathcal{H} = Pow(X)$. Consider a sequence $C = c_1 \dots c_m$ of context requirements with $c_i \in \mathcal{C}$, $i \in [1 : m]$. For switch $x \in X$ the relation $x \in h(\mathcal{C})$ holds, when $x \subset h$. During a reconfiguration operation the state of each available switch has to be defined. Hence, the costs of reconfiguration are simply the number of available reconfigurable units, i.e., $cost(h) = |h|$, where $|h|$ is the size of h, i.e. the number of switches available in h. Let $init(h) = w > 0$ for each $h \in \mathcal{H}$.

Thus during the execution of an algorithm/computation with sequence C of context requirements a machines performs operations $h_1 S_1 \dots h_r S_r$ where $h_1, \dots, h_r$ are hyperreconfigurations and S_i stands for a sequence of reconfigurations so that $C = S_1 \dots S_r$. The total reconfiguration time of a computation is measured as

$$r \cdot w + \sum_{i=1}^{r} |h_i| \cdot |S_i|$$

where $|S_i|$ is the length of S_i, i.e., the number of context requirements in S_i.

The problem to find for a given sequence $C = c_1 \dots c_m$ of context requirements a partition of C into substrings $S_1, \dots, S_r$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$, $r \geq 1$ such that $S_i \subset h_i(\mathcal{C})$ and $\sum_{i=1}^{r}(init(h_i) + cost(h_i) \cdot |S_i|)$ is minimized is called *Partition into Hypercontexts* (PHC) problem. In [LaMi04a] an optimal $O(\min\{n, m\} \cdot m^2)$ time algorithm for solving the PHC problem under the Switch model was given. It was also shown that the PHC problem under a more general cost measure becomes NP-complete.

3 A Flexible Control System

In this section we consider the problem of using a hyperreconfigurable machine for controlling processes that need a variable amount of supervision. We assume that several control tasks run on the machine for this purpose (compare Figure 1). Each task is given input data from the processes that it controls and it delivers output data to these processes. The reconfigurable machine is (hyper)reconfigured with (hyper)reconfiguration data — typically a stream of (hyper)reconfiguration bits — that are loaded from a host computer. We assume that there is a task selection process on the host computer which selects the control tasks that run on the reconfigurable machine. The task selection process gets control information from the processes that are controlled by the machine. The control information can indicate for example that other control tasks are needed than those that are currently running. It can also indicate that different control information is needed (which might lead to reconfiguration of the running tasks) or that the speed of feedback from the control process needs to be changed.

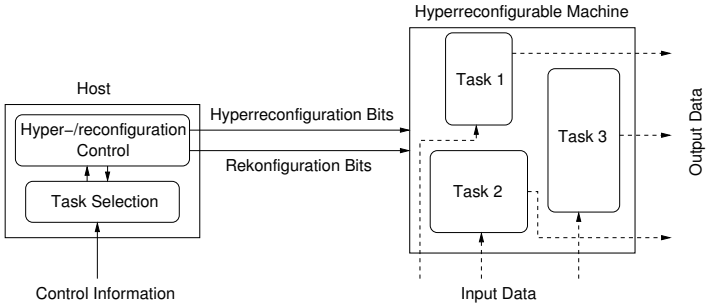

Figure 1: A hyperreconfigurable machine as a control system: general scheme and flow of information

Depending on the tasks selected, the hyper/reconfiguration control process computes the optimal times for hyperreconfiguration steps and the corresponding hypercontexts that

are used. Then it generates the hyperreconfiguration bitstreams and the reconfiguration bitstreams. These are then loaded onto the reconfigurable machine at the respective time steps. In some cases it might be necessary that the task selection process obtains a feedback on the (hyper)reconfiguration times that have been computed by the hyper/-reconfiguration control so that it can reconsider and possibly change its former task selection based on this information.

An interesting aspect of hyperreconfigurable machines is that the reconfiguration time for a set of tasks depends on the chosen hypercontext. This offers the possibility to change the run-time of a task without changing the task itself. One possibility is to exchange some of the other control tasks by variants that need less reconfiguration resources. Then other hypercontexts can be used under which altogether less reconfiguration information is needed for the tasks. This aspect of hyperreconfigurable architectures has not been studied before — in [LaMi04a, LaMi04b] only fixed task scenarios were considered.

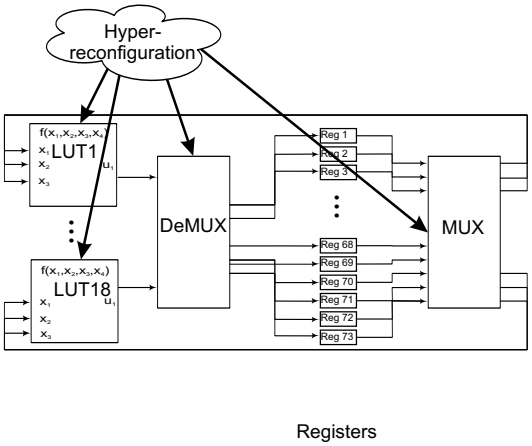

Figure 2: SHyRA architecture

4 The Example Architecture

An example architecture for a simple control system is described in this section. The machine is called the Simple HYperReconfigurable Architecture (SHyRA). As depicted in Figure 2 it has a set of 18 reconfigurable Look-Up Tables (LUTs) each of which has a 3-bit input and a 1-bit output port. For storing information a set of 73 registers is used. Different connections between the LUTs and the registers can be established over a reconfigurable 18:73 multiplexer and a 73:54 demultiplexer. The small number of LUTs poses a bottleneck for tasks and on this machine. Therefore, the control tasks used for the test runs have to make extensive use of reconfigurations.

Two types of simple control tasks are used for the test run. For one type of control task

Pseudo Code		Reconfiguration Data	
LUT1	LUT2	LUT1	LUT2
NOT 0, 0, 0	NOT 8, 0, 0	01010101	01010101
XOR 1, 1, 8	AND 8, 8, 1	01100110	00010001
XOR 2, 2, 8	AND 8, 8, 2	01100110	00010001
XOR 3, 3, 8	AND 8, 8, 3	01100110	00010001
EQ 9, 0, 4	ONE 8, 8, 0	10011001	11111111
EQ 9, 1, 5	AND 8, 8, 9	10011001	00010001
EQ 9, 2, 6	AND 8, 8, 9	10011001	00010001
EQ 9, 3, 7	AND 8, 8, 9	10011001	00010001
NULL 9, 0, 0	AND 8, 8, 9	00000000	00010001
CMOV 0, 9, 8	CMOV 1, 9, 8	01010011	01010011
CMOV 2, 9, 8	CMOV 3, 9, 8	01010011	01010011

Table 1: SHyRA 4 bit counter: pseudo code and corresponding reconfiguration bits for the two LUTs used

several variants have been implemented that differ in the number of LUTs they need and the number of computational cycles. One control task is a 4 bit binary counter with a variable upper bound. The counter increments its value that is stored in the registers one to four until it has reached the value that is stored in registers five to eight. The other control task is a 4 bit binary adder and the different variants use 9, 8, 7, 6, 5, or 4 LUTs respectively. The number of computational cycles that are used by these adders is 5, 5, 6, 7, 7, and 8 cycles respectively.

As all computational operations on SHyRA can only be performed through the use of LUTs neither the counter nor the adder can be implemented to run in one computational cycle. Hence, the design of these algorithms is time partitioned. As an example the operations done for the counter are shown in Table 1. The first two columns give a pseudo assembly code of the operations that are performed by the two LUTs. The notation used for an instruction is `OPCODE`, `OUT_REG`, `IN_REG1`, `IN_REG2`. The remaining two columns detail the reconfiguration data used to reconfigure the LUTs. Note, that in addition there is also reconfiguration information for the multiplexer and the demultiplexer. The two input signals used in the pseudo code are specified in the first two nibbles and the third is set to zero unless a `CMOV` operation is executed which requires the previous value of the output register as well. We analyzed the sequence of reconfigurations under the MT-Switch model where we assume that there are 48 switches (each corresponding to one of the 48 reconfiguration bits) that are local resources. For the multiple task model the tasks and the corresponding number of local switches are: $T_1 = LUT1$ with $l_1 = 8$, $T_2 = LUT2$ with $l_2 = 8$, $T_3 = DeMUX$ with $l_3 = 8$, and $T_4 = MUX$ with $l_4 = 24$.

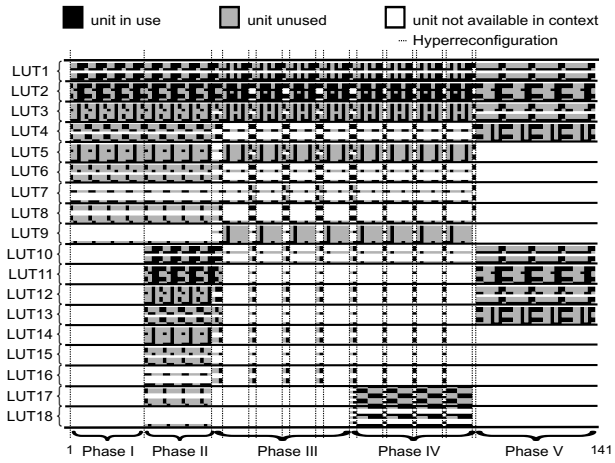

Figure 3: Phases with different control tasks on SHyRA: time of hyperreconfigurations state (available/not available + used/unused) of bits of the LUTs

5 Results

In the first experiment we study a situation where phases with different sets of control tasks run on the hyperreconfigurable machine. The tasks that run in the different phases are as follows. Phase I: 4 bit adder using 9 LUTs, Phase II: two 4-bit adders each using 9 LUTs, Phase III: 8-bit adder using 16 LUTs, Phase IV: 8-bit adder using 16 LUTs + 4-bit counter using 2 LUTs, Phase V: two 4-bit adders each using 4 LUTs. Altogether the phases comprise of 141 reconfiguration steps.

We first consider only the reconfiguration bits that are used to define the LUTs. This means that the cost for a hyperreconfiguration is 144 which equals the total number of reconfiguration bits for the 18 LUTs. For the determination of the best time steps for hyperreconfiguration and the selection of the corresponding hypercontext an optimal algorithm from [LaMi04a] was used. The results are depicted in Figure 3. The figure shows which bits of the LUTs are available in the hyperreconfiguration and which are actually used in each reconfiguration step. Figure 4 shows the total number of available and used bits for each reconfiguration step. Note, that between the borders of two phases a hyperreconfiguration was always done which is due to the different use of the reconfigurable resources. But also within phases II and III several hyperreconfigurations were done. It can be seen clearly that the number of bits/switches available for reconfiguration differ strongly between the phases and therefore leads to large saving of reconfiguration times in most phases.

When all reconfiguration bits are considered as hyperreconfigurable then the cost for a hyperreconfiguration equals the total number of reconfiguration bits for the LUTs and the multiplexer and demultiplexer. The total number of reconfigurable bits in this case is 5400. In one reconfiguration step only a few of the reconfiguration bits of the multiplexer and demultiplexer are used. This leads to high savings of reconfiguration bits due to hyperreconfiguration (less than 300 reconfiguration bits have to be specified in every re-

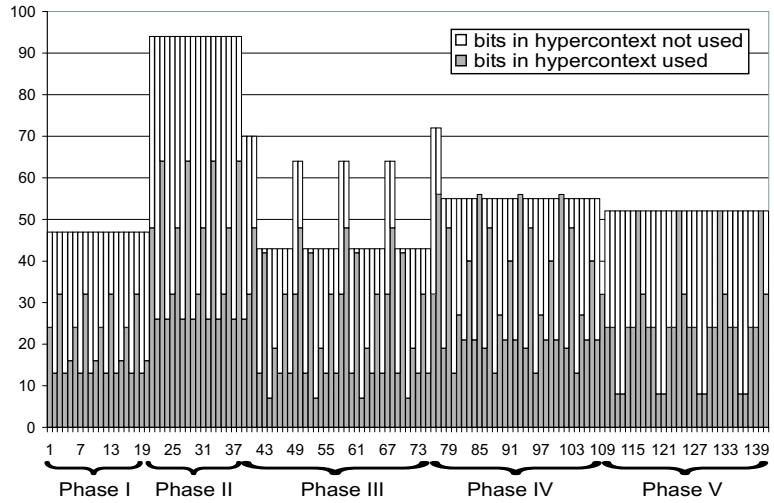

Figure 4: Phases with different control tasks on SHyRA: absolute number of available and used bits of the LUTs (only reconfiguration bits for LUTs are considered)

configuration). But the high costs for hyperreconfiguration lead to a much smaller number of hyperreconfigurations. Figure 5 shows the total number of available and used bits for each reconfiguration step.

The second experiment is designed to show an application scenario where the control information indicates that one control task should run faster whereas the running times of the other tasks are of minor importance. In such a case the task selection process might decide to use alternative versions of the latter which use less LUTs but may need more computational cycles. In the experiment we measure the change of (hyper)reconfiguration times during a run of the counter task when different adder tasks run in parallel with it.

	Adder					
	9 LUTs	8 LUTs	7 LUTs	6 LUTs	5 LUTs	4 LUTs
Bits for (hyper-) reconfigurations	7904	9344	8608	7552	7712	6336

Table 2: SHyRA with counter and different adders: total number of used (hyper)reconfiguration bits during 4 counter cycles

The counter task was run in parallel with each of the different 4-bit adder tasks. Each run was done over 4 counter cycles and the total number of (hyper)reconfiguration bits that have to be loaded during this time were measured. The results are shown in Table 2. The table shows that there is a clear tendency that the cost for (hyper)reconfiguration decreases when the counter use less reconfigurable resources, i.e. LUTs. This means that the counter tasks runs faster when the adder task is changed. In the experiment the (hyper)reconfiguration costs with a 4-bit adder is only about 68% of the cost when the

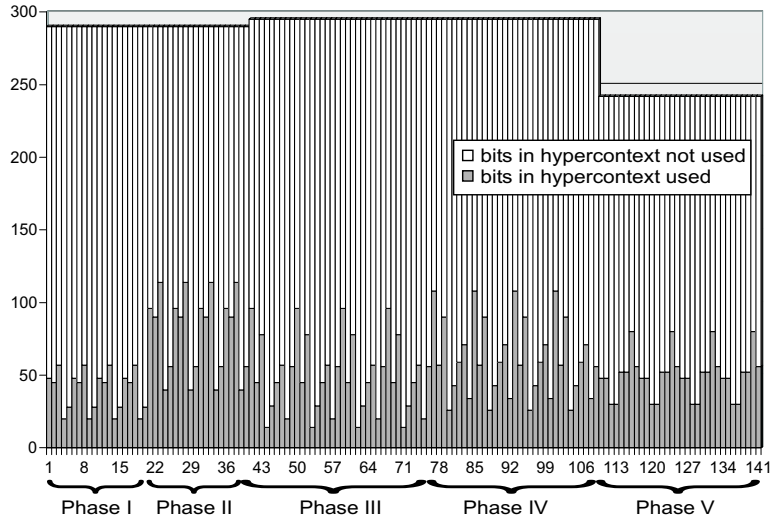

Figure 5: Phases with different control tasks on SHyRA: total number of available and used bits of the LUTs (all reconfiguration bits are considered)

8-bit adder runs in parallel. It is interesting that the (hyper)reconfiguration costs with the 9-bit adder are smaller than with the 8-bit and the 7-bit adder. The reason is that ideally the counter and the adder have to be designed together in order to obtain a minimal number of the (hyper)reconfiguration costs when run in parallel. This is because the reconfiguration bit usage of both tasks should fit in order to allow the use fewer and smaller hypercontexts. Since we cannot claim to have designed the optimal adders to fit with the counter it can happen that an adder which uses more LUTs fits better than an adder which uses less LUTs (e.g. the adders using 9 or 5 LUTs fit better than those using 8 respectively 4 LUTs).

6 Conclusion

We have shown in this paper that hyperreconfigurable architectures are a promising concept for the implementation of control tasks for processes with varying demands of supervision. In particular, the possibility to change the run-time of a task on such architectures without changing the task itself was investigated. It was shown how the amount of (hyper)reconfiguration that is necessary during the execution of a task can be changed by exchanging other tasks that run in parallel with variants that use less reconfiguration resources so that smaller hypercontexts can be used. An interesting topic for future research is how to design algorithms so that they fit well when they run in parallel on hyperreconfigurable architectures.

References

[DaPr01] A. Dandalis, V. K. Prasanna: Configuration Compression for FPGA-based Embedded Systems. In Proc. ACM International Symposium on Field-Programmable Gate Arrays, 173–182, (2001).

[HLR99] S. Hauck, Z. Li, and J.D.P. Rolim: Configuration Compression for the Xilinx XC6200 FPGA. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 8:1107–1113 (1999).

[KoeTe02] M. Koester, J. Teich: (Self-)reconfigurable Finite State Machines: Theory and Implementation. In Proc. 2002 Design, Automation and Test in Europe, 559–566 (2002).

[LaMi04a] S. Lange, M. Middendorf: Hyperreconfigurable Architectures and the Partition into Hypercontexts Problem. submitted (2003).

[LaMi04b] S. Lange, M. Middendorf: Models and Reconfiguration Problems for Multi-Task Hyperreconfigurable Architectures. Accepted for the 11th Reconfigurable Architectures Workshop (RAW 2004), Santa Fe, New Mexico (2004).

[LeWo02] K.K. Lee, D.F. Wong: Incremental Reconfiguration of Multi-FPGA Systems. In Proc. Tenth ACM International Symposium on Field Programmable Gate Arrays, 206–213 (2002).

[Midd03] M. Middendorf: Models and Architectures for Hyperreconfigurable Hardware (in German). Inaugural Meeting of the DFG Priority Programme 1148 (Reconfigurable Computing Systems), Stuttgart, 2003, manuscript.

[PaBu99] S.R. Park, W. Burleson: Configuration Cloning: Exploiting Regularity in Dynamic DSP Architectures. International Symposium on Field Programmable Gate Arrays (FPGA '99), 81-89, (1999).

[SWMP00] R.P.S. Sidhu, S. Wadhwa, A. Mei, and V.K. Prasanna: A Self-Reconfigurable Gate Array Architecture. Proc. FPL, Springer,LNCS 1896, 106–120 (2000).

[WaDa00] S. Wadhwa, A. Dandalis: Efficient Self-Reconfigurable Implementations Using On-chip Memory. Proc. FPL, Springer,LNCS 1896, 443–448 (2000).