Available online at www.sciencedirect.com

SCIENCE DIRECT®

J. Parallel Distrib. Comput. 65 (2005) 743 – 754

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Hyperreconfigurable architectures and the partition into hypercontexts problem☆

Sebastian Lange, Martin Middendorf*

Parallel Computing and Complex Systems Group, Department of Computer Science, University of Leipzig, Augustusplatz 10/11, D-04109 Leipzig, Germany

Received 2 August 2004; received in revised form 14 January 2005; accepted 23 January 2005
Available online 13 March 2005

Abstract

Dynamically reconfigurable architectures or systems are able to reconfigure their function and/or structure to suit the changing needs of a computation during run time. The increasing flexibility of modern dynamically reconfigurable systems improves their adaptability to computational needs but also makes fast reconfiguration difficult because of the large amount of reconfiguration information which has to be transferred. However, even when a computation uses this flexibility it will not use it all the time. Therefore, we propose to make the potential for reconfiguration itself reconfigurable. Such architectures are called hyperreconfigurable. Different models of hyperreconfigurable architectures are proposed in this paper. We also study a fundamental problem that emerges on such architectures, namely, to determine for a given computation when and how the potential for reconfiguration should be changed during run time so that the reconfiguration overhead is minimal. It is shown that the general problem is NP-hard but fast polynomial time algorithms are given to solve this problem for special types of hyperreconfigurable architectures. We define two example hyperreconfigurable architectures and illustrate the introduced concepts for corresponding application problems.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Dynamic reconfiguration; Reconfigurable architectures; Context partitioning; Reconfiguration costs

1. Introduction

Dynamically reconfigurable architectures or systems can adapt their function and/or structure to suit the changing needs of a computation during run time (e.g., [2,3]). A principle problem of dynamically reconfigurable systems is the tradeoff between flexibility and the amount of information needed for reconfiguration to define the new state of the system. Moreover, the increasingly higher integration of reconfigurable hardware, e.g. reconfigurable circuits on an FPGA chip, requires increased bandwidths for transferring the reconfiguration information. Modern FPGAs, for example, need several megabytes of reconfiguration data for a single reconfiguration step. This large amount of data transfer makes dynamic reconfigurations time critical operations, especially, for computations which exploit the full capacity of dynamically reconfigurable architectures by frequent reconfigurations.

Different approaches have been proposed in the literature to cope with this problem. Dandalis and Prasanna [4] have applied off-line compression methods to the stream of reconfiguration bits. The compressed stream of reconfiguration bits can be loaded faster onto the chip. Additional hardware is necessary on the chip which allows to decompress the reconfiguration bit stream during run time before it is needed to define the next configuration. Another method for compression of the reconfiguration bit stream which is suitable especially for the Xilinx XC6200 architecture has been described by Hauck et al. [7]. For Multi FPGA systems it has been proposed by Lee and Wong [11] to perform the reconfiguration incrementally so that only parts of the FPGAs

☆ 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".

* Corresponding author. Fax: +49 3419732329.
E-mail addresses: langes@informatik.uni-leipzig.de (S. Lange), middendorf@informatik.uni-leipzig.de (M. Middendorf).

need to be reconfigured at the same time. A third approach is to use self-reconfigurability which means that the reconfiguration bits are computed directly on the chip so that they can be transferred faster to the system units that are reconfigured (see Köster and Teich [9], Sidhu et al. [12], Wadhwa and Dandalis [16]). All these approaches have in common that they do not change the reconfiguration information itself.

In this paper, we propose a new approach to make run time reconfiguration faster by defining a new type of reconfigurable architectures. We use the fact that algorithms or computations typically consist of different phases where during each phase only a fraction of the reconfiguration potential of the underlying architecture is needed. The idea is to make the reconfiguration potential itself reconfigurable. The smaller the actual reconfiguration potential of an architecture is the smaller will the amount of reconfiguration information be that has to be transferred during reconfiguration and the faster will a reconfiguration step be. We call such architectures *hyperreconfigurable architectures*.

Hyperreconfigurable architectures use two types of reconfiguration steps: (i) reconfiguration steps where the reconfiguration potential of the architecture is defined (ii) standard reconfiguration steps which are used to reconfigure the hardware according to the contexts demanded by the algorithm. The first type of reconfiguration steps are called hyperreconfiguration steps.

A central problem that emerges on hyperreconfigurable architectures is to determine when hyperreconfiguration steps should be taken and how the reconfiguration potential should be defined in these steps in order to minimize the total time necessary for (hyper)reconfiguration of a computation. We call this problem Partition into Hypercontexts (PHC) problem and show that it is NP-hard. We also describe polynomial time algorithms for several variants of PHC on the so called Switch model and DAG model of hyperreconfigurable architectures. To illustrate the ideas in this paper we consider examples of two differently grained hyperreconfigurable architectures. For each of these architectures we study an instance of the PHC problem and give optimal solutions that were derived with the presented algorithms.

The paper is organized as follows. In Section 2, we describe the concept of hyperreconfigurable architectures. Formal models for such architectures and the Partition into Hypercontexts (PHC) problem are defined in Section 3. The NP-Hardness of PHC is shown in Section 4. In Sections 5 and 6 we discuss polynomial time solvable cases of the PHC problem. A variant of the PHC problem with changeover costs is studied in Section 7. Experimental results for the example architectures are presented in Section 8. The paper ends with a conclusion in Section 9.

2. The concept of hyperreconfigurable architectures

We call dynamically reconfigurable architectures and systems which allow to alter the reconfiguration potential during run time *hyperreconfigurable architectures*. Hyperreconfigurable architectures have two types of reconfiguration steps. The (*ordinary*) *reconfiguration steps* are used to actually define a new configuration of the system. The state of the system that can be changed by reconfiguration is called the *context* of a computation. *Hyperreconfiguration steps* are used for defining the actual reconfiguration potential of the architecture that is available for the ordinary reconfiguration steps. Thus, a hyperreconfiguration step defines the set of contexts that is available for the (ordinary) reconfiguration steps. Such a set of available contexts is called a *hypercontext*. With "available" we assign those reconfigurable resources that are activated by the hypercontext and therefore are available for reconfiguration. If a reconfiguration needs resources that are not included in the hypercontext they have to be activated/included by a hyperreconfiguration. We assume that a reconfiguration step requires reconfiguration information for all activated resources (even when the information is that an activated resource is not used in the corresponding context). Thus, we are interested in the case where the cost (e.g., the time or the amount of bits necessary to be loaded onto the architecture) of a reconfiguration step depends on the current hypercontext. Formal models for hyperreconfigurable architectures will be discussed in the next section.

This concept is illustrated in the following example. Consider a switch box for an FPGA where the state of each switch is determined by the content of a corresponding SRAM-cell as depicted on the right-hand side of Fig. 1. The content of these SRAM-cells is the current context of the switch box.

During reconfiguration the SRAM-cells are chained sequentially to form a shift register shifting in one bit of the new context and shifting out one bit of the old context at each time step. In order to enable hyperreconfigurability a second chain of SRAM cells is introduced to store the hypercontext of the switch box. Each of these cells manipulates two switches—one in front and one behind its corresponding (context)-SRAM-cell. These switches control whether the SRAM-cell is part of the switch register and the reconfiguration bits are sent through the SRAM-cell during reconfiguration or bypass it, thereby excluding it from reconfiguration. In the example of Fig. 1 two of the three shown hypercontext SRAM cells contain a 1 which means the corresponding context SRAM cells are included in the current chain of context SRAM cells. The other hypercontext SRAM cell contains a 0 which means the corresponding context SRAM cell is bypassed and therefore not included in the current chain of context SRAM cells. Since the time for reconfiguration is determined by the number of bits that have to be shifted in, it depends directly on the hypercontext. Loading a new hypercontext is done analogously to reconfiguration: the (hypercontext) SRAM-cells for a shift register and the bits of the new hypercontext are shifted in. Since the number of bits in the hypercontext is always the same, the time for a hyperreconfiguration step does not change. In this ex-

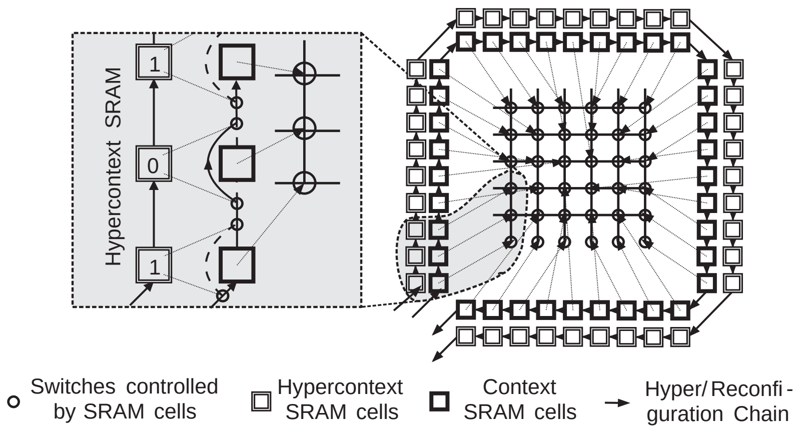
Hypercontext SRAM
1
0
1
Switches controlled by SRAM cells
Hypercontext SRAM cells
Context SRAM cells
Hyper/ Reconfiguration Chain

Fig. 1. Design example of a hyperreconfigurable switch box: the example hypercontext has one hypercontext SRAM cell set to 0 so that the corresponding context SRAM cell is bypassed for the context bits that are loaded into the chain of context SRAM cells.

ample it equals the time for a reconfiguration step where all SRAM-cells are included in the hypercontext.

Often it will not be possible to determine the context requirements of an algorithm exactly in advance. This is typically the case when a context depends on data that are computed at run time. For the concept of reconfigurable architectures it is enough when an upper bound on the requirements that will actually be needed during run time can be given. For example, it might be possible to know in advance that the routing requirements will be low during a certain phase of the algorithm even when the exact routing is not known in advance. Note, that several methods for resource estimation on reconfigurable architectures have appeared in the literature (e.g., see [8]).

3. Formal models and the partition into hypercontexts problem

In this section we introduce formal models for hyperreconfigurable architectures (or machines). The models which we introduce are general models that allow us to consider general algorithmic aspects for such architectures (and in particular the PHC problem). For concrete architectures these models can be made more specific.

We assume that an algorithm or a computation is characterized by a sequence of *context requirements*. Each context requirement describes the resource requirements that the algorithm/computation will have for a corresponding reconfiguration step that is performed during the run of the algorithm/computation. Hence, the number of context requirements equals the number of reconfiguration steps. Formally, let $\mathcal{C}$ be the set of possible context requirements for a reconfigurable machine. Then an algorithm/computation is characterized by the sequence

$$C = c_1 \dots c_m$$

of its context requirements during run time. Since the actual reconfiguration steps might depend on data that is only available at run time a context requirement always specifies the (estimated) maximal set of resources that could possibly be needed. 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 is reconfigured during the run of the algorithm.

A reconfiguration into a new context can in general only be realized during run time when the machine is in a hypercontext that contains at least all contexts possible according to the corresponding context requirement. In this case a hypercontext *satisfies* the corresponding context requirement. Formally, a *hypercontext* is a state of the reconfigurable machine which is characterized by the subset of $\mathcal{C}$ context requirements that are satisfied when the machine is in this state. At any time exactly one hypercontext is realized on the machine. Let $\mathcal{H}$ be the set of possible hypercontexts. 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 requirement c_i, $i \in [1:k]$ $c_i \in h(\mathcal{C})$ holds.

In the example hyperreconfigurable switch box (see Section 2) the set of hypercontexts $\mathcal{H}$ can be the set of all subsets of switches. A context requirement c can be a subset of the switches. Then for a hypercontext $h \in \mathcal{H}$ relation $c \in h(\mathcal{C})$ holds if $c \subset h$, i.e., all switches that are required for c are in the hypercontext h. The set of context requirements will usually depend not only on the architecture but also on the application and how good algorithm needs can be analyzed. For the switch box example the set of context requirements $\mathcal{C}$ can be the set of all subsets of switches. Then $\mathcal{H} = \mathcal{C}$. But it is also possible that $\mathcal{C}$ contains only a few subsets of switches, e.g., only the set of all switches and the set of all switches on the diagonal of the switch box.

In order to change the machine's current hypercontext a *hyperreconfiguration step* is necessary. To measure the costs for hyperreconfiguration steps and reconfiguration steps we introduce for each hypercontext $h \in \mathcal{H}$ two cost measures: (i) $init(h)$ is the cost of performing a hyperreconfiguration that brings the machine into hypercontext h (ii) $cost(h)$ denotes the cost of an ordinary reconfiguration step when the machine is in hypercontext h. In the example hyperreconfigurable switch box (see Section 2) $init(h)$ is the time to load the hypercontext bits into the hypercontext SRAM cells and $cost(h)$ is the time to load the context bits into the chain of context SRAM cells. Note that for this example $cost(h)$ depends on the current number of SRAM cells that are in the chain and are not bypassed.

A computation is characterized by a partition of C into substrings $S_1, \ldots, S_r$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$, $r \geqslant 1$ such that $S_i \subset h_i(\mathcal{C})$ and

$$\sum_{i=1}^{r} (init(h_i) + cost(h_i) \cdot |S_i|)$$

are the costs where $|S_i|$ is the length of S_i, i.e., the number of context requirements in S_i. When the algorithm/computation is executed the machine performs the following reconfiguration operations: $h_1 S_1 \ldots h_r S_r$ where S_i stands for a sequence of $|S_i|$ reconfigurations which use only those parts of the machine which are available within the hypercontext h_i. It is assumed that a hyperreconfiguration is always performed before the first reconfiguration step.

Fig. 2 shows an example of a computation on a hyperreconfigurable machine. Assume the machine contains 22 hyperreconfigurable resources $r_1, \ldots, r_{22}$, drawn as horizontal lines of boxes. The computation consists of a string $S = c_1 \ldots c_7$ of 7 context requirements. Two hyperreconfiguration operations partition S into two substrings $S_1 = c_1c_2c_3$ and $S_2 = c_4c_5c_6c_7$. During hyperreconfiguration resources are selectively enabled or disabled. Hypercontext h_1 disables resources $r_3, r_5, r_6, r_7, r_{14}, r_{15}, r_{16}, r_{18}$ and r_{19}. Observe, that all context requirements in substring S_1 do not specify reconfiguration data for this resources and thus are satisfied by h_1. Likewise hypercontext h_2 disables resources $r_1, r_2, r_4, r_6, r_{13}, r_{14}, r_{16}, r_{17}, r_{18}$ and r_{19} and satisfies all context requirements in substring $S_2 = \{c_4, c_5, c_6, c_7\}$.

An important problem that emerges for a hyperreconfigurable machine and a given algorithm (i.e. a sequence of context requirements) is to define when hyperreconfigurations are done and how corresponding hypercontexts are defined such that the context requirements of the algorithm are satisfied and the total costs for the hyperreconfiguration steps and the ordinary reconfiguration steps are minimized. The PHC problem can be defined as follows.

3.1. Partition into Hypercontexts (PHC) problem

Given a hyperreconfigurable machine (as described above) and a sequence $C = c_1 \ldots c_m$ of context requirements. Find a partition of C into substrings $S_1, \ldots, S_r$ (i.e., $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$, $r \geqslant 1$ with $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are minimal.

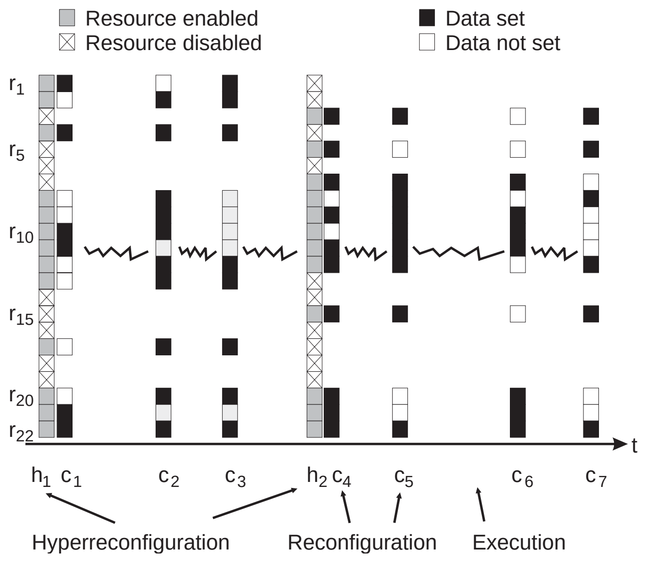

Fig. 2. Snapshot of a sample computation on a hyperreconfigurable machine with 7 context requirements $c_1 \ldots c_7$, where 2 hyperreconfigurations into hypercontexts h_1 and h_2 are done; computations within the respective current context are done between the reconfiguration operations; "data set" denotes the resources that are used within the context, "data not set" denotes the resources that are available within the current hypercontext but are not used by the context.

Two variants of the model for hyperreconfigurable architectures are introduced. The *DAG-model* is for coarse-grained reconfigurable machines where different reconfigurable submachines (hypercontexts) can be defined (through hyperreconfiguration) that can be ordered with respect to their computational power. It is assumed that a submachine which is an extension of another submachine and which has therefore a larger computational power produces more reconfiguration costs. This problem is typically used to model coarse-grained reconfigurable machines.

A directed acyclic graph (DAG) describes the precedence relation between the hypercontexts. Formally, given a DAG $G = (V, E)$ with $V = \mathcal{H}$ and for each $h \in \mathcal{H}$ a set $h(\mathcal{C})$ such that for each edge in $(h_1, h_2) \in E$ the relation $h_1(\mathcal{C}) \subset h_2(\mathcal{C})$ holds. It is assumed that a hypercontext h exists that satisfies all possible context requirements, i.e. $h(\mathcal{C}) = \mathcal{C}$. In addition let $cost(h) > 0$ and $init(h) = w$ for each $h \in \mathcal{H}$ and a constant $k \geqslant 0$ such that for each edge $(h_1, h_2) \in E$ $cost(h_1) \leqslant cost(h_2)$. Then a computation is characterized by a partition of C into substrings $S_1, \ldots, S_r$, $r \geqslant 1$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are

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

For each context requirement $c \in \mathcal{C}$ let $c(\mathcal{H})$ be the set of minimal (with respect to the precedence relation defined by

E) hypercontexts h in the DAG which satisfy $c \in h(\mathcal{C})$. The PHC problem for the DAG-model can be defined as follows.

3.2. PHC-DAG problem

Given a hyperreconfigurable machine in the DAG-model and a sequence $C = c_1 \dots c_m$ of context requirements. Find a partition of C into substrings $S_1, \dots, S_r$, $r \geqslant 1$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are minimal.

The second variant of a hyperreconfigurable machine is called *Switch-model*. In contrast to the DAG-model which is more confined to coarse-grained hyperreconfigurable machines the Switch-model is also well suited for fine-grained hyperreconfigurable machines. Here we assume that there exists a set of small (similar) reconfigurable units and every subset of these units can be used to define the reconfigurable machine that is available during a hypercontext. For example, each unit might be a switch and the set of switches defines the available part of the reconfigurable machine. An example are hyperreconfigurable switch boxes (see Section 2) of an FPGA which are used for connecting the functional units. The larger the (possible) routing requirements of an algorithm for a context, the more switches should be available in the hypercontext for reconfiguration during run time. For reconfiguration the state of each available switch has to be defined. Thus the cost for reconfiguration is just the number of available units plus some overhead cost. Hence, for this Switch model the reconfigurable machine that is available during a hypercontext is defined by the subset of available units.

Formally, let $X = \{x_1, \dots, x_n\}$ be a set of switches and define $\mathcal{C} = \mathcal{H} = 2^X$, i.e., the set of possible context requirements $\mathcal{C}$ and the set of possible hypercontexts $\mathcal{H}$ equal the set of all subsets of X. For context $x \in X$ the relation $x \in h(\mathcal{C})$ holds, when $x \subset h$. Let $cost(h) = |h|$, where $|h|$ is the size of h, i.e., the number of switches available in h. Let $init(h) = n$ for $h \in \mathcal{H}$, which reflects the fact that for each switch it has to be defined during hyperreconfiguration whether it is available in the new hypercontext. A computation is characterized by a partition of C into substrings $S_1, \dots, S_r$, $r \geqslant 1$ (i.e., $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are

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

The PHC problem for the Switch-model can be defined as follows.

3.3. PHC-Switch problem

Given a hyperreconfigurable machine in the Switch-model with set of switches $X = \{x_1, \dots, x_n\}$ and a sequence of context requirements $C = c_1 \dots c_m$. Find a partition of C into substrings $S_1, \dots, S_r$, $r \geqslant 1$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and the total (hyper)reconfiguration costs are minimal.

Note that for the PHC-Switch problem there exist 2^n hypercontexts but this number is not part of the size of the problem instance which is $n + m$. This is different for the PHC-DAG where the DAG is part of the instance and therefore the number of possible hypercontexts is also part of the instance.

In order to solve the PHC problem we make a simple but useful observation. A partial order $\preccurlyeq$ on the set of hypercontexts (i.e., $\preccurlyeq$ is a reflexive, antisymmetric, and transitive relation on $\mathcal{H}$) can be defined naturally by the subset relation on the sets of contexts that are satisfied by the hypercontexts. Thus, $h_1 \preccurlyeq h_2$ iff $h_1(\mathcal{C}) \subset h_2(\mathcal{C})$ and $h_1 \prec h_2$ iff $h_1(\mathcal{C}) \subset h_2(\mathcal{C}) \wedge h_1(\mathcal{C}) \neq h_2(\mathcal{C})$. The partial order $\preccurlyeq$ is called *cost consistent*, when for each two hypercontexts h_1, $h_2 \in \mathcal{H}$ with $h_1 \prec h_2$ it follows that $init(h_1) \leqslant init(h_2)$ and $cost(h_1) \leqslant cost(h_2)$ and $<$ holds in at least one case.

Observation. If $\preccurlyeq$ is cost consistent then for a solution of PHC, i.e., a partition of C into substrings $S_1, \dots, S_r$ and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$, each hypercontexts h_i, $i \in [1 : r]$ is minimal within the set of all hypercontexts $h \in \mathcal{H}$ with $S_i \subset h(\mathcal{C})$ (i.e., there exists no hypercontext $h \in \mathcal{H}$, $h \neq h_i$ with $S_i \subset h(\mathcal{C})$ and $h \prec h_i$).

It is easy to see that relation $\preccurlyeq$ is cost consistent for the DAG-model and the Switch-model.

4. NP-Hardness

In this section we state that the general PHC problem is NP-complete, which means it is unlikely that the problem can be solved in polynomial time.

Theorem 1. *The PHC problem is NP-complete.*

We only give the proof idea because the proof itself is somewhat technical and uses standard constructions for proving NP-completeness. For a proof one can encode an instance of an NP-hard problem, say 3-SAT, in a sequence of contexts C. Then a cost function and a set of hypercontexts can be defined such that there exists a cheap partition into hypercontexts of C if and only if the partition consists of a single hypercontext and the contexts in C encode an instance of 3-SAT that is solvable such that there exists no partition of C into substrings which can be covered by hypercontexts in a cheap way.

5. Polynomial time algorithm for PHC-DAG

The algorithm for the PHC-DAG problem which is described in this section is a dynamic programming algorithm that computes a table $M = (M_{k,j})_{k \in [1:m], j \in [k:m]}$ where $M_{k,j}$

are the minimal costs for the prefix of length j of the sequence of context requirements $c_1 \dots c_m$ when using k hypercontexts. The optimal solution for the PHC-DAG model can then be derived with standard dynamic programming techniques from this matrix.

In the following let h_{ij} be a cheapest hypercontext that satisfies the contexts requirements $c_i, \dots, c_j$. The algorithm for the PHC-DAG problem consists of the following steps:

(1) Preprocessing: For each $i, j \in [1 : m]$, $i < j$ cost $cost(h_{ij})$ is computed.
(2) Initialization: Every element in first row of the matrix is determined, i.e. $M_{1,j} := w + cost(h_{1j}) \cdot j$ for $j \in [1 : m]$.
(3) Computation of $M_{k,j}$ for $k \in [2 : m]$, $j \in [k : m]$ according to

$$M_{k,j} = \min\{M_{k-1,p-1} + w + cost(h_{p,j}) \cdot (j - p - 1) \mid p \in [k : j]\}.$$

(4) Computation of the quality of the optimal solution from the matrix M by determining

$$\min\{M_{k,p} \mid k \in [1 : m]\}.$$

Run time analysis: The preprocessing step takes time $O(m^2 \cdot \alpha)$ where α is the cost of computing the set of all minimal hypercontexts from two sets of hypercontexts in the DAG that are predecessor of at least one hypercontext in both sets and then to determine the cheapest of these hypercontexts as well. Initialization takes time $O(m \cdot \alpha)$ since each element can be determined in time $O(\alpha)$. For step (3) the algorithm considers an element $M_{k,j}$ with $k > 1$ and assumes that all elements in row $k - 1$ have already been determined. Then it is clear that the computation of a single element in (3) takes time at most time $O(j)$. Step (iv) takes time $O(m)$. Hence we can derive the following theorem:

Theorem 2. *The PHC-DAG problem can be solved in time* $O(m^3 + \alpha \cdot m^2)$.

6. Polynomial time algorithm for PHC-Switch

In this section we describe a dynamic programming solution for the PHC-Switch problem. The algorithm computes a table $M = (M_{k,j})_{k\in[1:m], j\in[k:m]}$ where $M_{k,j}$ are the minimal costs for the prefix of length j of the sequence of context requirements $c_1 \dots c_m$ when using k hypercontexts. The optimal solution for PHC-Switch can then be derived from this matrix. This algorithm is designed such that each row of the matrix can be determined in time $O(n \cdot m)$ so that the total run time is in $O(n \cdot m^2)$.

In the following let h_{ij} be a cheapest hypercontext that satisfies the contexts requirements $c_i, \dots, c_j$. First, we need some facts and definitions. It is not hard to show for each $k \in [1 : m]$: (i) the value of $M_{k,p}$ is monotone increasing in p, and (ii) for $j \in [k : m]$ the value of $cost(h_{i,j})$ is monotone decreasing in i. Let $j \in [k : m]$. It follows from the stated facts that there exists a partition $T_1, \dots, T_h$ of the sequence of context requirements $c_k \dots c_j$ such that $c_k \dots c_j = T_1 \dots T_h$ and for each string of contexts T_s, $s \in [1 : h]$ holds: For all contexts $c_t \in T_s$ the hypercontexts $h_{t,j}$ and therefore the costs $cost(h_{t,j})$ are the same. Recall, that $h_{t,j}$ for the PHC-Switch problem is defined as the hypercontext that consists of all switches that are element of at least one of the context requirements $c_t, \dots, c_j$, i.e., $h_{t,j} = \bigcup_{i=t}^{j} c_i$. We call the partition $T_1, \dots, T_h$ the *equal cost partition* of $[k : j]$. The corresponding intervals of indices of the contexts the *equal cost intervals*.

Let $[s : t]$ be an equal cost interval. For index $x \in [s : t]$ the values $\delta \in [1 : n]$ are determined for which $M_{k,x-1} + \delta \cdot (t - (x - 1)) = \min\{M_{k,y-1} + \delta \cdot (t - (y - 1)) \mid y \in [s : t]\}$ holds. Clearly, for each index $x \in [s : t]$ the corresponding δ values form a subinterval of $[1 : n]$. This interval is called the *minimum cost interval of index x* (within the equal cost interval $[s : t]$) and is denoted by I_x. It is not hard to show that $I_s, \dots, I_t$ is a partition of $[1 : n]$ where all elements in I_i are smaller than all elements in I_{i+1} for $i \in [s : t - 1]$.

In the following we describe the computation of a single matrix element in the main step of the algorithm. We assume that all elements in row 1 of $M_{k,j}$ and all elements $M_{k,k} = k \cdot w + \sum_{i=1}^{k} |c_i|$, $k \in [1 : m]$ have been computed during initialization. It is enough to consider the computation of an element $M_{k,j+1}$ for $k > 1$ and $j \in [1 : m - 1]$ assuming that elements in row $k - 1$ and element $M_{k,j}$ have already been computed.

In order to search efficiently for possible good places to introduce the kth hyperreconfiguration we introduce a pointer structure over parts of the sequence of context requirements $c_1 \dots c_m$. First we describe the pointer structure over the sequence $c_k \dots c_j$ for the computation of $M_{k,j}$ and then show how it can be extended to a pointer structure over the sequence $c_k \dots c_{j+1}$ for the computation of $M_{k,j+1}$.

The first context requirements in each of the sequences of context requirements $T_h, \dots, T_1$ are linked by so called *equal cost pointers*, i.e. there is a pointer to the first context requirement in T_h, from there to the first context requirement in T_{h-1} and so forth. Moreover, within each equal cost interval the indices x with a minimal cost interval that is empty or contains only values that are smaller than the actual costs $cost(h_{x,j+1})$ are linked in order of increasing value by so-called *minimum cost pointers*. In addition, there is a pointer from the first context requirement of the interval to the last useful index in the interval. This pointer is called the *end pointer* of the equal cost interval. All indices with an equal cost interval that are linked by minimal cost pointers are called *useful*. All other indices are called *useless* and will be marked as useless by the algorithm. The following two facts which are not hard to show are used for run time analysis and to show the correctness of the algorithm.

Fact 1. It is easy to obtain the equal cost partition $U_1 \dots U_g$ of $c_k \dots c_{j+1}$ of $[k : j + 1]$ and the corresponding pointers from the equal cost partition $T_1, \dots, T_h$ of $[k : j]$ and its corresponding pointers in time $O(n)$.

To see that this is true observe that each string in $U_1, \dots, U_g$ can be obtained by merging (or copying) neighbored strings from $T_1, \dots, T_h$ and U_g contains in addition the context requirement c_{j+1}.

Fact 2. Consider an element T_s of the equal cost partition $T_1, \dots, T_h$ of $[k : j]$. Let c_x (c_y) be the context in T_s (respectively from the element of the equal cost partition of $[k : j+1]$ that contains T_s) for which $M_{k,x-1} + cost(h_{x,j})$ (respectively $M_{k,y-1} + cost(h_{y,j+1})$) is minimal. Then it follows that $x \leqslant y$.

To compute $M_{k,j+1}$ the algorithm performs the following steps:

(i) Extend the equal cost partition of $[k : j]$ by appending the (preliminary) equal cost interval c_{j+1} and let $[1 : n]$ be the (preliminary) minimal cost interval for $j+1$.

(ii) Compute the equal cost partition of $[k : j+1]$ from the extended equal cost partition of $[k : j]$ by merging neighbored intervals when they have the same cost with respect to $j+1$.

(iii) For each index within a merged interval the new equal cost interval is determined together with its minimal cost pointers and its end pointer. During this process all indices that have become useless are marked.

Clearly step (i) can be done in time $O(1)$. The determination of the intervals that have the same costs in step (ii) is done in time $O(n)$ by following pointers that connect the intervals. To determine the time for step (iii) consider an equal cost interval $[s_0 : s_h]$, $k \leqslant s_1 \leqslant s_h \leqslant j+1$ that was merged from $h \leqslant n$ old intervals $[s_0 : s_1]$, $[s_1+1 : s_2], \dots, [s_{h-1}+1 : s_h]$. We now show that the computation of new pointers and the marking of useless indices takes time $O(h+q)$ where q is the number of marked indices.

(a) For every of the h intervals consider the minimum cost interval of the index to which the first minimum cost pointer points. If the minimum cost interval does not contain a value that is at least as large as $cost(h_{s,j+1})$ then the index is marked as useless and the first pointer is merged with the next pointer. This process proceeds until every first minimum cost pointer points to a useful index.

(b) Now it remains to update the minimum cost intervals by selecting for each cost value only the best index from the h merged intervals. This can be done in a left to right manner starting with the smaller cost values. Thereby always comparing the corresponding minimum cost intervals of indices between two neighbored of the h merged intervals, say $[s_{i-1}+1 : s_i]$ and $[s_i+1 : s_{i+1}]$, $i \in [1 : h-1]$. For ease of description we assume here that all values in one minimal cost interval are better than all values in the other interval. If this is not the case both minimum cost intervals are split so that each contains only the values for which it is better. Observe that the split value can be computed in constant time. When the minimum cost interval in the left interval $[s_{i-1}+1 : s_i]$ is better the corresponding index in the right interval is marked useless and the next minimum cost intervals are compared. When the minimum cost interval in the right interval $[s_i+1 : s_{i+1}]$ is better the index in the left interval is marked useless. Then the minimum cost interval in the old right interval (now the new left interval) is compared with the corresponding minimum cost interval of its right neighbor interval $[s_{i+1}+1 : s_{i+2}]$. During the search for the corresponding minimum cost interval all indices that are passed are marked useless. The process stops when the best minimum cost interval with value n is found. During the search a pointer is set from the rightmost useful index of an interval to the first useful index in its right neighbor. Thereby it might be necessary to jump over intervals that have no useful index left. The end pointer of the first interval is set to point to the last useful index of the merged intervals.

Since the total number of intervals in the equal cost partition for $[k : j+1]$ is at most n minus the number of merged intervals the time to compute $M_{k,j+1}$ is at most $O(n+q)$ where q is the total number of indices that are marked useless. Since at most $m-k$ indices exist in row k of matrix M it follows that the computation sum of all steps (iii) for computing the elements in this row is $O(n \cdot m + m)$.

Theorem 3. *The PHC-Switch problem can be solved in time* $O(n \cdot m^2)$.

7. PHC with changeover costs

In this section we study a variant of the PHC problem where the cost for a hyperreconfiguration depends not only on the new hypercontext but also on the predecessor hypercontext. Parts of the hyperreconfiguration costs can then be considered as changeover costs and therefore we call this problem the PHC problem with changeover costs. This problem is used to model architectures where during hyperreconfiguration it is not necessary to specify the new hypercontext from scratch but where it is possible to define the new hypercontext through its difference to the old hypercontext.

For this problem the changeover costs between two hypercontexts are defined as the number of switches for which the state has to be changed for the new hypercontext (i.e., the state is changed from available to not available or vice versa). Formally, the problem is defined as follows.

PHC-Switch with changeover costs problem: Given an instance of the PHC-Switch problem, where $init(h) = w$ for $h \in \mathcal{H}$, $w > 0$, the cost function *changeover* on $\mathcal{H} \times \mathcal{H}$ is defined by $changeover(h_1, h_2) := |h_1 \triangle h_2|$ where $\triangle$ denotes the symmetric difference, and an initial hypercontext $h_0 \in \mathcal{H}$. Find a partition of C into substrings $S_1, \dots, S_r$, $r \geqslant 1$ (i.e. $C = S_1 \dots S_r$) and hypercontexts $h_1, \dots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and $r \cdot w + \sum_{i=1}^{r}(|h_i \triangle h_{i+1}| + |h_i| \cdot |S_i|)$ is minimized.

The PHC-Switch with changeover costs problem is more difficult to solve than the PHC-Switch problem because the costs of a hypercontext depends on the predecessor hypercontext. A consequence is that $\preccurlyeq$ is not cost consistent for this model and therefore it is not enough to consider only minimal hypercontexts in the sense as defined in Section 3. In the following we describe a polynomial time dynamic programming algorithm for the PHC-Switch with changeover costs problem. Due to the cost model for hyperreconfiguration it is not advantageous to remove and add a switch x from/to the hypercontext when less than 3 contexts not using x are between the corresponding hyperreconfigurations. Therefore, we assume that for each switch x a subsequence of maximal length of context requirements $c_i, \ldots, c_j$ which all do not use x has length at least 3. It is easy to guarantee in time $O(m \cdot n)$ that this property holds. For ease of description we assume that at least 5 hyperreconfigurations are done (including the hyperreconfiguration that is employed before the first context).

The algorithm computes a table $M = (M_{k,l,j,z})$, $k \in [5 : m-1]$, $l \in [k : m]$, $z \in \{0, 1, 2\}$, $j < l-1$ for $z = 0$ and $j < l-2$ for $z \in \{1, 2\}$ where $M_{k,l,j,z}$ are the minimal costs for the prefix of length $l-1$ of the sequence of contexts when using k hypercontexts and

- for $z = 0$ a hyperreconfiguration is done at l and the last hyperreconfiguration before c_{l-1} is at $j < l-1$,
- for $z = 1$ hyperreconfigurations are done at l and $l-1$, and the last hyperreconfiguration before c_{l-2} is at $j < l-2$,
- for $z = 2$ hyperreconfigurations are done at l, $l-1$, $l-2$ and the last hyperreconfiguration that is done before c_{l-3} is at $j < l-2$,

where in all three cases the minimal costs include only cost w for the hyperreconfiguration at l but not the costs for removing or adding switches from/to the hypercontext during this hyperreconfiguration. The reason is that when computing $M_{k,l,j,z}$ it is only decided that a hyperreconfiguration is done at l but not how the hypercontext is defined. This hypercontext is determined and the corresponding costs for adding/removing switches are counted when the $k+1$th hypercontext is introduced or when the final costs for the reconfigurations up to the last context are determined.

It is easy to see that all values $M_{k,l,j,z}$ in the matrix with $k = 5$ can be computed in time $O(m^3 \cdot n)$ similarly as the values with $k \geqslant 6$ as it is described in the following: We describe only the main step of the algorithm to compute $M_{k,l,j,z}$ when all values $M_{k-1,.,.,.}$ have already been computed.

The algorithm needs a preprocessing step where for each $i \in [1 : m]$ a sorted list of all switches that are not used at step i is created. The list is called the list of non-used switches of i and is denoted $list(i)$. For each element $x \in list(i)$ the index $v_i(x)$ of the first context after context c_i when x is used again is computed. Also, the index $u_i(x)$ of the last context where x was used before context c_i is computed. Clearly this preprocessing step can be done in time $O(m \cdot n)$. For the description of the main step of the algorithm we make the following case differentiation depending on the value of z.

(1) *Case* $z = 0$: $M_{k,l,j,0}$ with $j < l-1$ is the minimum over all the following cost values:
  (i) for each $M(k-1, j, h, 0)$ with $h < j-1$ the cost $M(k-1, j, h, 0)$
    + w,
    + the number of switches $x \in list(j-1)$ that have to be included additionally at the $k-1$th hyperreconfiguration at j because they are used in a context c_{l-1} or earlier (i.e., $v_{j-1}(x) < l$) and that are not included in the hypercontext at h (because $u_{j-1}(x) < h$),
    + the number of switches $y \in list(j)$ that have to be removed from the hypercontext at j because they are included in the hypercontext at h (because $h \leqslant u_j(y)$) and that are not used again before c_l (i.e., $l \leqslant v_j(y)$),
    + the cost for reconfiguration of the contexts $c_j, \ldots, c_{l-1}$, i.e., the number of switches that are (now) in the hypercontext at j times $(l-j)$,
  (ii) for each $M(k-1, j, h, 1)$ and each $M(k-1, j, h, 2)$ with $h < j-2$ the cost $M(k-1, j, h, 1)$ (respectively, $M(k-1, j, h, 2)$)
    + w,
    + the number of switches $x \in list(j-1)$ that have to be included in the hypercontext additionally at j with $v_{j-1}(x) < l$ and $u_{j-1}(x) < h$ (respectively $u_{j-1}(x) < j-2$) (similar as in case (i) but see the following remark),
    + the number of switches $y \in list(j)$ that have to be removed from the hypercontext at j (because $v_j(x) \geqslant l$) and are not included in the hypercontext at $j-1$ (because $u_j(y) = j-1$),
    + the number of switches that are (now) in the hypercontext at j times $(l-j)$.

Remark. Observe that for $M(k-1, j, h, 1)$ ($M(k-1, j, h, 2)$) switches $x \in list(j-1)$ with $h \leqslant u_{j-1}(x) < j-1$ (respectively, $u_{j-1}(x) = j-2$) have been removed from the hypercontext at the hyperreconfiguration at $j-1$. When for such a switch $v_{j-1}(x) < l$ then it has to be included again during the hyperreconfiguration at j. But since it is cheaper not to perform these two changes of the state of switch x we assume x has not been removed at the hyperreconfiguration at $j-1$. Note, that this does not change the correct value of $M(k-1, j, h, 1)$ (respectively, $M(k-1, j, h, 2)$).

(2) *Case* $z = 1$: $M_{k,l,j,1}$ with $j < l-2$ is the minimum over all the following cost values: for $M(k-1, l-1, j, 0)$ the cost $M(k-1, l-1, j, 0)$
  + w,
  + the number of switches $x \in list(l-2)$ that have to be included additionally at $l-1$ because $v_{l-2}(x) = l-1$) and $u_{l-2}(x) < j$ (similar as in (1.i)),

+ the number of switches $y \in list(l-1)$ that have to be removed from the hypercontext at $l-1$ (because $v_{l-1}(y) \geqslant l$ and $j \leqslant u_{l-1}(y)$ (similar as in (1.i)),
+ the number of switches that are (now) in the hypercontext at $l-1$.

(3) *Case* $z = 2$: $M_{k,l,j,2}$ with $j < l-2$ is the minimum over all the following cost values: for $M(k-1, l-1, j, 1)$ or $M(k-1, l-1, h, 2)$ with $h < l-3$ the cost $M(k-1, l-1, j, 1)$ (respectively, $M(k-1, l-1, h, 2)$)
+ w,
+ the number of switches $x \in list(l-2)$ that have to be included additionally at $l-1$ because $v_{l-2}(x) = l-1$) and $u_{l-2}(x) < j$ (respectively, $u_{l-2}(x) < l-3$) (similar as in (1.i)) but see the following remark),
+ the number of switches $y \in list(l-1)$ that have to be removed from the hypercontext at $l-1$ (because $v_{l-1}(y) \geqslant l$ and $l-2 = u_{l-1}(y)$ (similar as in (1.i)),
+ the number of switches that are (now) in the hypercontext at $l-1$.

Remark. Note that each switch $x \in list(j-1)$ with $v_{l-2}(x) = l-1$ has $u_{j-1}(x) < l-4$. Then observe that switches $x \in list(j-1)$ with $j \leqslant u_{j-1}(x) < l-2$ have been removed from the hypercontext at the hyperreconfiguration at $l-2$ (considering $M(k-2, l-2, j, 0)$). When for such a switch $v_{l-2}(x) = l-1$ then it has to be included again during the hyperreconfiguration at $l-1$. But since it is cheaper not to perform this two changes of the state of switch x we assume x has no been removed at the hyperreconfiguration at $l-1$ (respectively, $l-3$). Note, that this does not change the correct value of $M(k-1, l-1, j, 1)$.

To find the optimal solution for the PHC-Switch problem with changeover costs from the table $(M_{k,l,j,z})$ one has to compute for each element in the table the total costs for hyperreconfigurations and reconfigurations. For element $M_{k,l,j,z}$ one has to determine which switches have to be removed/included into the hypercontext at l (similarly as has above). Then the number of this switches plus $M_{k,l,j,z}$ plus the costs for the configurations of contexts $c_l, \ldots, c_m$ are the total costs. The minimum over all total costs gives the costs of the optimal solution. It is not hard to determine also the optimal solution using standard techniques for dynamic programming. Since the computation of one element $M_{k,l,j,z}$ takes time at most $O(m^2 \cdot n)$, there are $O(m^2)$ elements in the matrix, and the preprocessing takes time $O(m \cdot n)$ we have shown the following theorem:

Theorem 4. *The PHC-Switch with changeover costs problem can be solved in time* $O(m^4 \cdot n)$.

For the PHC-DAG with changeover costs problem we assume that for each pair of hypercontexts h_1, h_2 changeover costs *changeover*(h_1, h_2) are defined. Formally, the problem is defined as follows.

PHC-DAG with changeover costs problem: Given a hyperreconfigurable machine in the DAG-model where $init(h) = w$ for $h \in \mathcal{H}$, $w > 0$, a nonnegative cost function *changeover* on $\mathcal{H} \times \mathcal{H}$ and a sequence $C = c_1 \ldots c_m$ of context requirements. Find a partition of C into substrings $S_1, \ldots, S_r$, $r \geqslant 1$ (i.e. $C = S_1 \ldots S_r$) and hypercontexts $h_1, \ldots, h_r$ such that $S_i \subset h_i(\mathcal{C})$ and $r \cdot w + \sum_{i=1}^{r}(changeover(h_i, h_{i+1}) + |h_i| \cdot |S_i|)$ is minimized.

Theorem 5. *The PHC-DAG problem with changeover costs problem can be solved in time* $O(m^3 \cdot |\mathcal{H}|^2)$.

The proof is omitted because the problem can be solved with similar dynamic programming techniques as shown above.

8. Experiments and results

In order to illustrate the concept of hyperreconfiguration and to make a basic evaluation of the models and algorithms presented above two differently grained hyperreconfigurable architectures are considered in this section. As an example of a simple model of a fine-grained dynamically reconfigurable machine we define the Simple HYperReconfigurable Architecture (SHyRA) (see Fig. 3). It consists of a set of reconfigurable Look-Up Tables (LUTs) each with three inputs and one output. For storing signals a file of registers is used, which are reconfigurably connected to the LUTs by a multiplexer (MUX) and a demultiplexer (DeMUX). The system interfaces to the outside world through the content of the register file. The execution of a computation on SHyRA is cycle based. Each computational cycle consists of the following two steps: (i) Input stimuli are propagated from the registers through the multiplexer to the corresponding inputs of the LUTs, (ii) a new output is generated in the LUTs and stored through the demultiplexer into the register file. Hyper-/Reconfiguration is possible before every computational cycle. Due to the architecture's inability to directly chain the LUTs for computation in one computational cycle application processes are forced to make extensive use of (hyper)reconfigurations. The Switch-model of hyperreconfigurable architectures fits to the SHyRA architecture because each reconfiguration bit for each of the three types of reconfigurable resources can be viewed as a binary switch.

As a test application for the SHyRA we have chosen a part of a simple control task consisting of a sequence of 5 computation phases with counter and adder circuits. After mapping the design onto the reconfigurable resources (LUTs and MUX/DeMUX) the sequence of context requirements which is depicted in Fig. 4 was determined by analyzing the change of the reconfiguration bits at each clock cycle.

For a more coarse-grained example a VLIW processor is seen as a hyperreconfigurable machine where the functional units are the reconfigurable resources. A reconfiguration step takes place every clock cycle. Executing an algorithm on

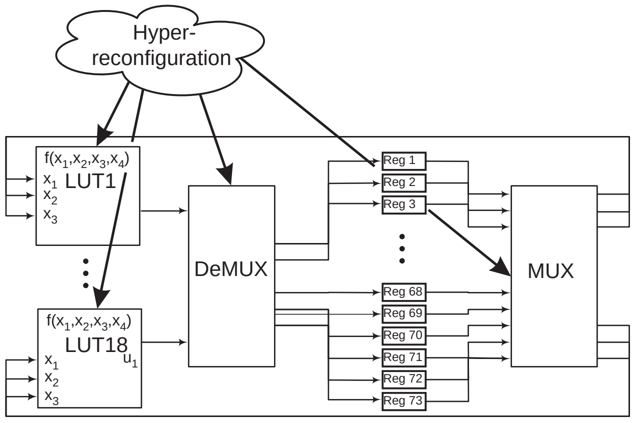

Fig. 3. Simple HYperReconfigurable Architecture: principal system design.

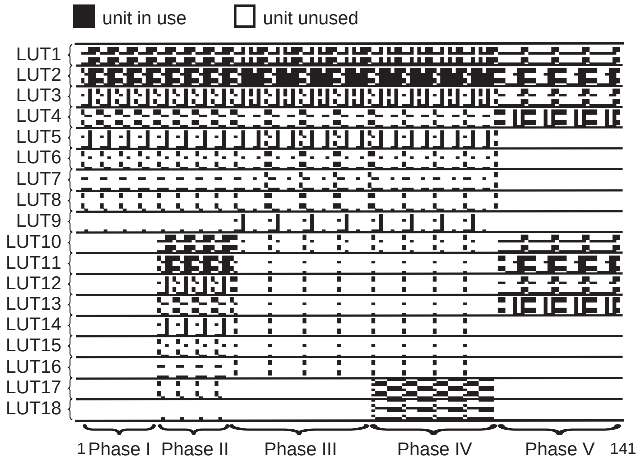

Fig. 4. Sequence of 141 context requirements for control task application on SHyRA: For each context requirement the status (required/used or not required/used) of each of 8 bits for all 19 LUTs is given.

a VLIW processor thus closely resembles a rapidly reconfiguring system. Because of the high number of functional units a VLIW code word rather frequently encodes several idle operations for units currently not usable due to a lack of parallelism in the algorithm. The concept of hyperreconfigurability can be employed to limit the effect of these idle operations.

The TMS320C6201 signal processor from Texas Instruments was used for the experiments. It features 4 types of functional units (L $=$ logic, M $=$ multiply, D $=$ load/store, S $=$ shift) with 2 units each totalling in 8 functional units. The accompanying C compiler produces optimized and parallelized code and outputs the assembly code as well, which was used for our simulations. As an example algorithm a vector summation was implemented in C, compiled into parallelized VLIW code and executed. During execution the state (idle or busy) of each functional unit was recorded for each clock cycle, yielding an 8-bit vector for every clock cycle. The sequence of these vectors for the VLIW processor can be seen as a sequence of

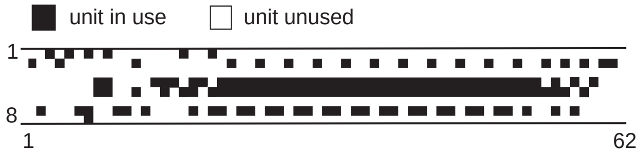

Fig. 5. Sequence of 62 context requirements for the vector summation application on VLIW processor: For each context requirement the status (required/used or not required/used) of each of 8 functional units is given.

context requirements (see Fig. 5) for a hyperreconfigurable machine that works in the Switch-model. The test run with a vector size of 44 resulted in 62 executed VLIW instructions, corresponding to 62 context requirements.

The obtained sequences of context requirements for the fine-grained SHyRA and the coarse-grained VLIW can be seen as instances of the PHC-Switch problem. For both problem instances the optimal times for hyperreconfigurations and the corresponding hypercontexts were determined with the algorithm from Section 6 for standard costs and with the algorithm from Section 7 for changeover costs. Both test instances were evaluated with different init costs for the hyperreconfiguration. For the standard model the costs were $init(h) = n + k$ where n is the total number of bits ($n = 144$ for the SHyRA and $n = 8$ for the VLIW) and $k \in [0, 150]$ can be seen as additional base costs. For the changeover cost model $init(h) = |h_1 \triangle h_2| + k$ where h_1, h_2 are the context requirements before and after the hyperreconfiguration and $k \in [0, 150]$ can be seen as the base costs.

Fig. 6 shows the optimal total (hyper)reconfiguration costs for different base costs $k \in [0, 150]$ for the control task on SHyRA. The figure shows also the total number of hyperreconfigurations. It can be observed that the costs for the PHC-Switch model with changeover costs are significantly lower than in the standard PHC-Switch model for the same base costs. The reason is that the changeover model can profit from the fact that for this application consecutive hypercontexts are often similar. Therefore, the PHC-Switch model with changeover costs can profit more from the hyperreconfigurations in the sense that it does more hyperreconfigurations in the optimal solution. Especially, when the base costs are low the number of hyperreconfigurations is much higher as for the standard cost model.

The optimal costs and number of hyperreconfigurations for vector summation on the VLIW are shown in Fig. 7. Here again the standard PHC-Switch model produces significantly higher costs when the base costs are small. But different to the SHyRA application the number of hyperreconfigurations is often the same for both cost models. For higher base costs the total hyperreconfiguration costs for both cost models become very similar because in both cases only one hypercontexts is used. Thus, the total hyperreconfiguration costs differ only by the cost for the first hyperreconfiguration (this is $n + k$ for standard costs and $n + k$ minus the number of bits that are never used for changeover costs).

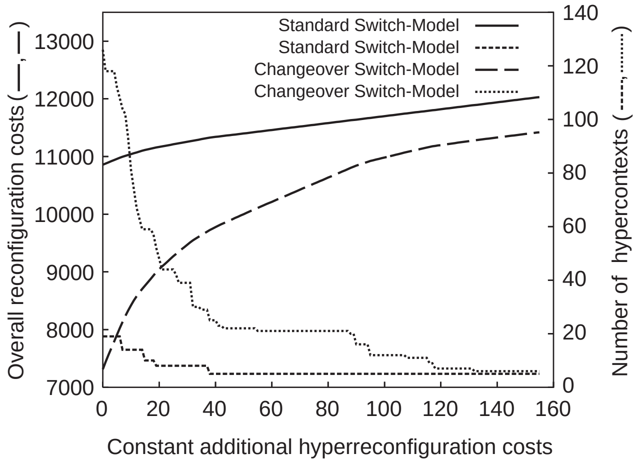

Fig. 6. Total (hyper)reconfiguration costs and number of hypercontexts with changing hyperreconfiguration signaling costs for a control task on SHyRA.

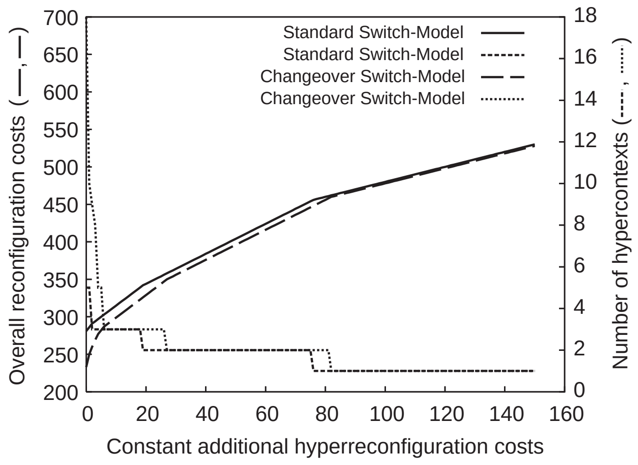

Fig. 7. Total (hyper)reconfiguration costs and number of hypercontexts with changing hyperreconfiguration signaling costs for a vector summation on a VLIW machine.

In order to evaluate the possible profit from hyperreconfigurations we compared the total (hyper)reconfiguration costs for either of the two hyperreconfiguration models with the case that hyperreconfigurations are not allowed. For the latter case it is assumed that all switches have to be fully specified at each reconfiguration step. Fig. 8 shows the relative total hyperreconfiguration costs for the hyperreconfigurable case (100% are the total reconfiguration costs when hyperreconfigurations are not allowed). For the PHC-Switch model with changeover costs and base costs zero the total (hyper)reconfiguration costs are reduced to 36% for the control task and 53% for the vector summation example. Even the standard cost model reduced the costs to 47% and, respectively, 57% for base costs zero. Clearly, when the base costs are increased the reduction decreases. In the case of the vector summation the break-even point where it becomes

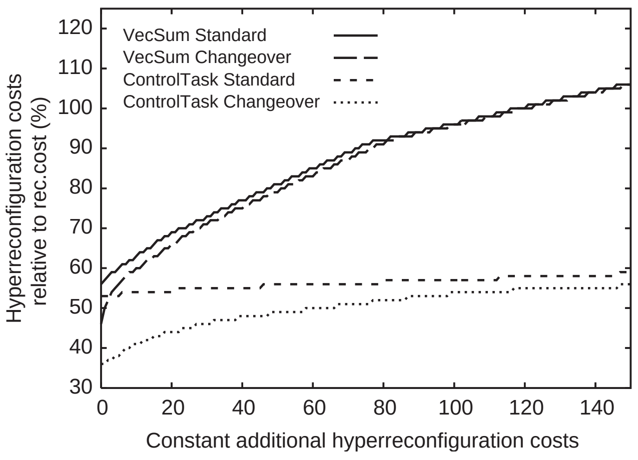

Fig. 8. Ratio of total (hyper)reconfiguration costs for the hyperreconfigurable SHyRA and VLIW applications compared to the total reconfiguration cost for the case that no hyperreconfigurations are allowed.

equally expensive to employ the concept of hyperreconfiguration appears with base costs of 118 for the standard model and 120 for the model with changeover costs. These are extreme cases where the size of a hypercontext (and thus its actual costs for specification) is 8 and thus the additionally introduced base costs are 14.75 times larger. For the control task instance hyperreconfiguration reduced the costs by more than 40% even with base costs 150. The results show that there is a large potential for increasing the speed of dynamic reconfigurations when using hyperreconfigurations. Of course, the cost model which is used here is only a simple example to illustrate our approach and it has be to figured out for real architectures how hyperreconfiguration can exactly be realized.

9. Conclusion

The concept of hyperreconfigurable architectures was introduced in this paper. This was motivated by the observation that for dynamically reconfigurable architectures or systems there is a tradeoff between flexibility and the amount of reconfiguration information that has to be transferred to the reconfigurable units. Hyperreconfigurable architectures offer a new type of reconfiguration operations—called hyperreconfigurations—that allow to alter their potential for reconfiguration that is available during run time. A central problem that emerges on hyperreconfigurable systems is to determine for a given computation the best points in time when a hyperreconfiguration should be performed. This problem has been called the Partition into Hypercontexts (PHC) problem. It was shown that the general PHC problem is NP-complete. But for several models of hyperreconfigurable machines fast polynomial time algorithms have been described to solve the PHC problem. As an illustrating example and to evaluate the introduced concepts we presented solutions of the PHC problem for example algorithms on a fine-grained and a coarse-grained reconfigurable architecture. The concept of hyperreconfigurations offers interesting questions for further research. For instance how can such architectures be realized and what are the best cost measures? How can the resource requirements be estimated? What specific problems occur for multi task hyperreconfigurable architectures? While we have considered the latter question in a companion paper [10] the other questions will be subject to our future research.

References

[2] K. Bondalapati, V.K. Prasanna, Reconfigurable Computing: Architectures, Models and Algorithms, in: Proceedings of the Reconfigurable Architectures Workshop, 1997, .

[3] K. Compton, S. Hauck, Configurable computing: a survey of systems and software, ACM Comput. Surveys 34 (2) (1994) 171–210.

[4] A. Dandalis, V.K. Prasanna, Configuration compression for FPGA-based embedded systems, in: Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays, 2001, pp. 173–182.

[7] S. Hauck, Z. Li, J.D.P. Rolim, Configuration compression for the Xilinx XC6200 FPGA, IEEE Trans. on CAD of Integrated Circuits and Systems (8) (1999) 1107–1113.

[8] P. Kannan, S. Balachandran, D. Bhatia, On metrics for comparing routability estimation methods for FPGAs, in: Proceedings of the 39th Design Automation Conference, 2002, pp. 70–75.

[9] M. Koester, J. Teich, (Self)-reconfigurable finite state machines: theory and implementation, in: Proceedings of the 2002 Design, Automation and Test in Europe, 2002, pp. 559–566.

[10] S. Lange, M. Middendorf, Multi task hyperreconfigurable architectures: models and reconfiguration problems, Internat. J. Embedded Systems, to appear (Preliminary version in Proceedings of the 11th Reconfigurable Architectures Workshop, 2004).

[11] K.K. Lee, D.F. Wong, Incremental reconfiguration of multi-FPGA systems, in: Proceedings of the 10th ACM International Symposium on Field Programmable Gate Arrays, 2002, pp. 206–213.

[12] R.P.S. Sidhu, S. Wadhwa, A. Mei, V.K. Prasanna, A self-reconfigurable gate array architecture, in: Proceedings of the FPL, 2000, pp. 106–120.

[16] S. Wadhwa, A. Dandalis, Efficient self-reconfigurable implementations using on-chip memory, in: Proceedings of the FPL, 2000, pp. 443–448.

Martin Middendorf received the Diploma degree in Mathematics and a Dr. rer. nat. at the University of Hannover, Germany, in 1988 and 1992, respectively. He gained his professorial Habilitation in 1998 at the University of Karlsruhe, Germany. He has worked at the University of Dortmund, Germany, and the University of Hannover, Germany, as a visiting Professor of Computer Science. He was Professor of Computer Science at the Catholic University of Eichstätt, Germany. Currently he is professor for Parallel Computing and Complex Systems with the University of Leipzig, Germany. His research interests include reconfigurable architectures, parallel algorithms, algorithms from nature and bioinformatics.

Sebastian Lange received the Diploma degree in Computer Science at the University of Leipzig, Germany, in 2003. He started his Ph.D. at the Department of Computer Science, University of Leipzig, in the same year. Currently he is working on Hyperreconfigurable Architectures within the DFG Priority Programme Reconfigurable Computing Systems. His research interests include reconfigurable architectures, fault-tolerant design and bio-inspired algorithms.