The Partition into Hypercontexts Problem for Hyperreconfigurable Architectures

Sebastian Lange and Martin Middendorf

Parallel Computing and Complex Systems Group
Department of Computer Science, University of Leipzig
Augustusplatz 10/11, D-04109 Leipzig, Germany
{langes,middendorf}@informatik.uni-leipzig.de

Abstract. Hyperreconfigurable architectures adapt their reconfiguration abilities during run time in order to achieve fast dynamic reconfiguration. Models for such architectures have been proposed that change their ability for reconfiguration during hyperreconfiguration steps and in ordinary reconfiguration steps reconfigure the actual contexts for a computation within the limits that have been set by the last hyperreconfiguration step. In this paper we study algorithmic aspects of how to optimally decide what hyperreconfiguration steps should be done during a computation in order to minimize the total time necessary for hyperreconfiguration and ordinary reconfiguration. It is shown that the general problem is NP-hard but fast polynomial time algorithms are given to solve this problem on different types of hyperreconfigurable architectures. These include newly introduced architectures that use a cache to store hypercontexts. We define an example hyperreconfigurable architecture and illustrate the introduced concepts for three application problems.

1 Introduction

The increasingly higher integration and flexibility of dynamically reconfigurable hardware lead to a large amount of information which has to be transferred onto the hardware for reconfiguration to define the new state of the system. This large amount of data transfer makes run time 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, e.g., compression methods for the stream of reconfiguration bits ([4,6]), multi-context architectures [1,12]), self-reconfigurability ([8,15,17]) and hyperreconfiguration ([9]) which means that the reconfiguration potential of an architecture itself is reconfigurable.

In this paper we study algorithmic aspects of single task hyperreconfigurable architectures as they have been proposed in [9] (algorithmic aspects of multi-task hyperreconfigurable architectures are studied in [10]). Such 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 actual contexts which are used by the algorithm. The first type of reconfiguration steps are called hyperreconfiguration steps. Moreover, we extend hyperreconfigurable architectures by introducing a cache for storing hypercontexts.

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 of hyperreconfigurable architectures ([9]). Unfortunately, it is also shown that the introduction of a cache for hypercontexts makes the PHC problem NP-hard even for the Switch model. To illustrate the ideas in this paper we present an example for the PHC problem on the Switch model. An optimal solution for the PHC problem is provided when the example architecture has no cache and a heuristic solution when a cache for hypercontexts is used.

The paper is organized as follows. In the next Section 2 we describe hyperreconfigurable architectures and introduce the Partition into Hypercontexts (PHC) problem. In Section 4 we discuss polynomial time solvable cases of the PHC problem. A variant of the PHC problem with changeover costs is studied in Section 5. In section 6 we introduce hyperreconfigurable architectures with a cache for hypercontexts and study PHC for these architectures. Experimental results for a test architecture are presented in Section 7. The paper ends with a conclusion in Section 8.

2 The Partition into Hypercontexts Problem

Hyperreconfigurable architectures allow to alter the reconfiguration potential during run time and use two types of reconfiguration steps ([9]). The ordinary reconfiguration steps are used to actually define a new configuration of the system. The actual 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 activated for reconfiguration in the ordinary reconfiguration steps. Thus, a hyperreconfiguration step defines the set of contexts that can potentially be reconfigured in (ordinary) reconfiguration steps. Such a set of possible contexts is called a hypercontext. A reconfiguration into a new context might be dependent on external and internal parameters of the computation and can be characterized by the set of all possible contexts that it defines depending on the data. Hence, a reconfiguration can in general only be executed during run time when the machine is in a hypercontext that contains this set of possible contexts. A set of possible contexts is called a context requirement and a hypercontexts that contains it *satisfies* the corresponding context requirement. It is assumed 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). Formal models for hyperreconfigurable architectures 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 actual hypercontext have been given in [9] and are described in the following.

Let $\mathcal{C}$ be the set of possible context requirements for a reconfigurable machine and $C = c_1 \ldots c_m$, $c_i \in \mathcal{C}$ be the sequence of context requirements that characterizes an algorithm/computation. A *hypercontext* is a state of the 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 $c_1 \ldots c_k$ of context requirements and a hypercontext h let $c_1 \ldots c_k \subset h(\mathcal{C})$ denote the fact that for each context requirement c_i, $i \in [1:k]$ $c_i \in h(\mathcal{C})$ holds. In order to change the machine's current hypercontext a *hyperreconfiguration step* is necessary. For each hypercontext $h \in \mathcal{H}$ two cost measures are defined: 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. Then 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 $h_1, \ldots, h_r$, $r \geq 1$ such that $S_i \subset h_i(\mathcal{C})$ and $\sum_{i=1}^{r}(init(h_i) + cost(h_i) \cdot |S_i|)$ 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.

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. Formally we define,

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 $h_1, \ldots, h_r$, $r \geq 1$ swith $S_i \subset h_i(\mathcal{C})$ and minimal total (hyper)reconfiguration.

Two variants of the model for hyperreconfigurable architectures have been introduced in [9]. The *DAG model* is for coarse grained reconfigurable machines where different reconfigurable submachines (hypercontexts) can be defined that can be ordered with respect to their computational power (this model is not considered in this paper due to space limitations). The second variant called *Switch model* is for fine grained machines where a set of small (similar) reconfigurable units (also called switches) exists. The reconfigurable machine that is available during a hypercontext is defined by the subset of available units. For reconfiguration the state of each available switch has to be defined. Thus the cost for reconfiguration is the number of available units plus some overhead cost. Formally, let $X = \{x_1, \ldots, x_n\}$ 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 $S_1,\dots,S_r$, $r \geq 1$ (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 $r \cdot n + \sum_{i=1}^{r} |h_i| \cdot |S_i|$.

PHC-Switch problem: Given a hyperreconfigurable machine in the Switch model with the 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 $S_1,\dots,S_r$, $r \geq 1$ (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 $n+m$.

3 NP-Hardness

In this section we show that the general PHC problem is NP-hard 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. 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.

4 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 $O(n \cdot m^2)$.

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 decreasing in p, 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 therefor 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 $cM_{k,x-1} + \delta \cdot (t-(x-1)) = \min\{M_{k,y-1} + \delta \cdot (t-(y-1)) \mid y \in [s:$

$t]\}$ 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 $I_s, \ldots, I_t$ 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 $c_1 \ldots c_m$. First we describe the pointer structure over the sequence $c_k \ldots c_j$ for the computation of $M_{k,j}$ and then show how it can be extended to a pointer structure over the sequence $c_k \ldots c_{j+1}$ for the computation of $M_{k,j+1}$.

The first context requirements in each of the sequences of context requirements $T_h, \ldots, T_1$ 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 (omitted due to space limitations).

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

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

Fact 2: Consider an element T_s of the equal cost partition $T_1, \ldots, T_h$ 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 $x \leq y$.

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 $[s_0 : s_h]$, $k \leq s_1 \leq s_h \leq j+1$ that was merged from $h \leq n$ old intervals $[s_0 : s_1], [s_1+1 : s_2], \ldots [s_{h-1}+1 : s_h]$. 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 each 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 2. *The PHC-Switch problem can be solved in time $O(n \cdot m^2)$.*

5 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 its preceding 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. In the following we consider the problem only for the Switch-Model. 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 can be stated as follows.

PHC-Switch problem with changeover costs: 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 $S_1, \ldots, S_r$, $r \geq 1$ (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 $r \cdot w + \sum_{i=1}^{r} (|h_i \triangle h_{i+1}| + |h_i| \cdot |S_i|)$ is minimized.

The next result shows that PHC-Switch with changeover costs is polynomially solvable (the algorithm is too involved for the available space and omitted).

Theorem 3. *The PHC-Switch problem with changeover costs can be solved in time* $O(m^4 \cdot n)$.

6 Caches for Hypercontext and PHC

Multi-context devices allow to store the reconfiguration data that are necessary to specify a set of contexts. Such context caching on the device can lead to a significant speedup compared to single context devices where the reconfiguration bits have to be loaded onto the device from a host computer for every reconfiguration. In this section we introduce multi-hypercontext hyperreconfigurable architectures, which have a cache for hypercontexts so that they can switch between hypercontexts very rapidly. The concept of reconfigurable devices with context switching has been introduced a decade ago (e.g. the dynamically configurable gate array (DPGA) [1] or WASMII [12]). In [14] the reconfigurable computing module board (RCM) has been investigated which contains two context-switching FPGAs, called CSRC, where the context switching device can store four contexts.

A typical cache problem for many reconfigurable architectures is that the sequence of contexts for a computation is known in advance and the problem is then to find the best replacement strategies for the contexts that are stored in the cache. On a run time reconfigurable machine the problem is that the actual contexts might not be known in advance because they can depend on the actual results of a computation. But what might be known in advance are general requirements on the contexts, e.g. whether few or many routing resources are needed. The actual context, e.g. the exact routing, is then defined at a reconfiguration step. Therefore, it seems a promising concept for hyperreconfigurable architectures to introduce a cache for storing hypercontexts.

What makes the problem of using a cache for hypercontexts particularly interesting on a hyperreconfigurable machine is that different sequences of hypercontexts are possible which can satisfy the sequence of context requirements of a computation. Hence, the algorithm that computes the best sequence of hypercontexts should take the use of the cache into account. In general, it can be advantageous to use fewer but more comprehensive hypercontexts in order to increase the chances that a hypercontext which is to be used already exists in the cache and can therefore be loaded very fast. Thus, there is a trade-off between the increasing reconfiguration costs when fewer but more comprehensive hypercontexts are used and the shrinking costs for loading these hyperreconfigurations.

Here we consider a hyperreconfigurable machine with a cache for hypercontexts that can store a fixed maximal number of hypercontexts. It is assumed that a hypercontext has to be loaded from the host only when the hypercontext is not stored in the cache. Hence, the cost for loading a hypercontext h depends on whether it is in the cache or not. The value of $\mathit{init}(h)$ is smaller when the hypercontext is in the cache. For a machine with cache we define the PHC-Switch problem as follows.

PHC-Switch problem (for hyperreconfigurable machines with a cache for hypercontexts): Given a cache capacity $2n$, a set of switches $X = \{x_1, \ldots, x_n\}$, a set of context requirements $\mathcal{C}$ and a set of hypercontexts $\mathcal{H}$ defined as $\mathcal{C} = \mathcal{H} = 2^X$,i.e., $\mathcal{C}$ and $\mathcal{H}$ equal the set of all subsets of X. For a given sequence of context requirements $C = c_1 \ldots c_m$ find a partition of C into substrings $S_1, \ldots, S_r$, $r \geq 1$ (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 $r_1 \cdot n + r_2 \cdot c + \sum_{i=1}^{r} |h_i| \cdot |S_i|$ is minimized where r_2 is the number of hypercontexts that can be loaded from the cache, $r_2 := r - r_1$, and c the cost to load a hypercontext from the cache.

We can show the following theorem by a reduction from 3-SAT (the proof is somewhat technical and therefore omitted).

Theorem 4. *The PHC-Switch problem is NP-hard on a hyperreconfigurable machine with a cache for hypercontexts.*

7 Experiments and Results

We define a Simple HYperReconfigurable Architecture (SHyRA) as an example of a minimalistic model of a rapidly reconfiguring machine in order to illustrate our concepts. As depicted in Figure 1 it features 18 reconfigurable Look-Up Tables each with three inputs and one output. For storing signals a file of 73 registers is used. The registers are reconfigurably connected to the LUTs by a 73:54 multiplexer and 18:73 demultiplexer. The inability of the architecture to directly chain the LUTs for computation poses a bottle neck for the test applications we run on SHyRA and forces them to make extensive use of reconfigurations. The test applications therefore naturally lend themselves to profit from the use of hyperreconfigurations. This, however, does not limit the general validity of the experimental results, because although SHyRA implicitly imposes reconfiguration every reconfigurable application follows the same basic design, i.e. having a calculation phase (LUTs), transferring the information to some registers (DeMUX) and then have it reinjected into the next calculation phase (MUX). In order to evaluate the caching model, each reconfigurable component was equipped with a cache of up to 14 cache lines. Two sample applications (a 4 bit adder and a primitive ALU) were mapped to the modified SHyRA.

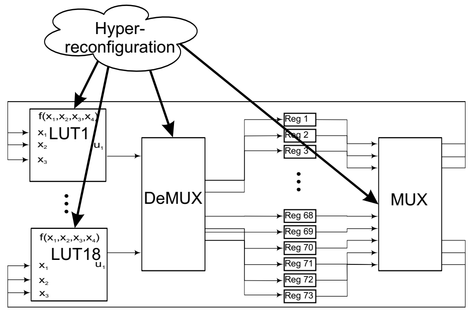

Fig. 1. Simple HYperReconfigurable Architecture: Principal System Design

After mapping the design onto the reconfigurable resources (LUT contents, MUX switching information) a heuristic was employed to determine appropriate hypercontexts using the same costs as in the Switch model. For the case of not using caches the optimal hypercontexts were determined with the algorithm described in Section 4. For the case with caches for hypercontexts we used a greedy strategy which takes the optimal solution for the PHC-Switch problem without caches as starting point and subsequently improves this solution by randomly applying one of three operations:

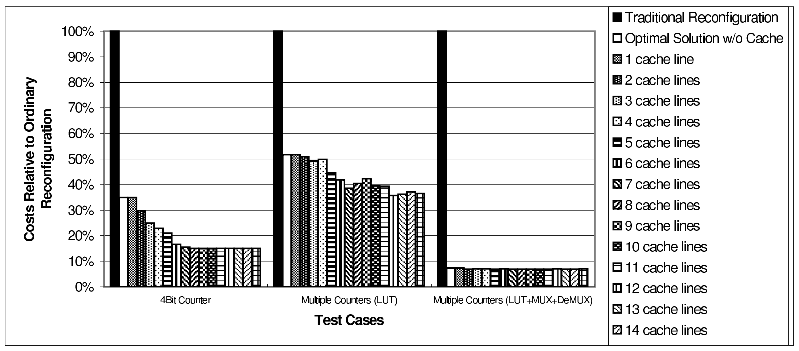

Fig. 2. Relative Costs of the Test Case Designs With Cache Size From 1 to 14 Lines

1. Two randomly chosen hypercontexts are merged. 2. Two hypercontexts are chosen randomly. For each context c_j a penalty cost $(cost(c_j) = \sum_{k\in[0,n],c_{jk}=1}(|\{c_i | i \neq j, c_{ik} = 0\}|))$ is determined and the most expensive context is exchanged (this is repeated as long as the total costs become smaller). 3. One randomly chosen hypercontext is split into two hypercontexts and the same exchange procedure as in (2) is applied.

Figure 2 shows the resulting total hyperreconfiguration costs for the test designs without cache and with caches of sizes from one two 14 cache lines. For the test applications it can be observed that small caches for hypercontexts can significantly decrease the total hyperreconfiguration costs.

8 Conclusion

We have investigated a central algorithmic problem for hyperreconfigurable architectures, namely the Partition into Hypercontexts (PHC) problem. It was shown that the problem in NP-hard in general but can be solved in polynomial time for the Switch model under different cost measures. We have also introduced hyperreconfigurable architectures that use a cache to store hypercontexts and have shown that PHC becomes NP-hard even for the Switch model for this architectures. Applications of the PHC problem on an example architecture have been given. For the case when caches for hypercontexts are used a heuristic for solving the PHC problem was introduced.

References

1. M. Bolotski, A. DeHon, and Jr. T.F. Knight: Unifying FPGAs and SIMD Arrays. Proc. FPGA '94 – 2nd International ACM/SIGDA Workshop on FPGAs, 1-10, (1994).
2. K. Bondalapati, V.K. Prasanna: Reconfigurable Computing: Architectures, Models and Algorithms. In Proc. Reconfigurable Architectures Workshop, IPPS, (1997).
3. K. Compton, S. Hauck: Configurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 34(2): 171–210, (2002).
4. A. Dandalis and V. K. Prasanna: Configuration Compression for FPGA-based Embedded Systems. In Proc. ACM Int. Symposium on Field-Programmable Gate Arrays, 173–182, (2001).
5. C. Haubelt, J. Teich, K. Richter, and R. Ernst: System Design for Flexibility. In Proc. 2002 Design, Automation and Test in Europe, 854–861, (2002).
6. S. Hauck, Z. Li, and J.D.P. Rolim: Configuration Compression for the Xilinx XC6200 FPGA. IEEE Trans. on CAD of Integrated Circuits and Systems, 8:1107–1113, (1999).
7. P. Kannan, S. Balachandran, D. Bhatia: On Metrics for Comparing Routability Estimation Methods for FPGAs. In Proc. 39th Design Automation Conference, 70–75, (2002).
8. M. Koester and J. Teich: (Self-)reconfigurable Finite State Machines: Theory and Implementation. In Proc. 2002 Design, Automation and Test in Europe, 559–566, (2002).
9. S. Lange and M. Middendorf: Hyperreconfigurable Architectures for Fast Runtime Reconfiguration. To appear in Proceedings of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04), Napa Valley, USA, 2004.
10. S. Lange and M. Middendorf: Models and Reconfiguration Problems for Multi Task Hyperreconfigurable Architectures. To appear in Proc. RAW 2004, Santa Fe, 2004.
11. K.K. Lee and D.F. Wong: Incremental Reconfiguration of Multi-FPGA Systems. In Proc. Tenth ACM International Symposium on Field Programmable Gate Arrays, 206–213 , (2002).
12. X. P. Ling, and H. Amano: WASMII: a Data Driven Computer on a Virtual Hardware. Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, 33-42, (1993).
13. T.-M. Lee, and J. Henkel, W. Wolf: Dynamic Runtime Re-Scheduling Allowing Multiple Implementations of a Task for Platform-Based Designs. In Proc. 2002 Design, Automation and Test in Europe, 296–301, (2002).
14. K. Puttegowda, D.I. Lehn, J.H. Park, P. Athanas, and M. Jones: Context Switching in a Run-Time Reconfigurable System. The Journal of Supercomputing, 26(3): 239-257,(2003).
15. R.P.S. Sidhu, S. Wadhwa, A. Mei, V.K. Prasanna: A Self-Reconfigurable Gate Array Architecture. Proc. FPL (2000) 106-120.
16. M. Teich, S. Fekete, and J. Schepers: Compile-Time Optimization of Dynamic Hardware Reconfigurations. Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, U.S.A., 1999.
17. S. Wadhwa, A. Dandalis: Efficient Self-Reconfigurable Implementations Using On-chip Memory. Proc. FPL, (2000) 443-448.