

Cache Architectures for Reconfigurable Hardware

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—The architecture and use of caches for two-level reconfigurable hardware is studied in this paper. The considered two-level reconfigurable hardware performs ordinary reconfiguration operations at the lower reconfiguration level. Whereas the upper reconfiguration level is used to configure the capabilities that are actually available for the lower level. The actual state of each reconfiguration level is determined by a corresponding context. The use of context caches and strategies for their use in ordinary 1-level reconfigurable architectures have been studied several times in the literature. Here we propose different architectures for caches which can store lower and upper level contexts for 2-level reconfigurable hardware. In addition we propose several heuristics that reduce the total reconfiguration costs when using upper level cache. Experimental results for fine grained and coarse grained two-level reconfigurable architectures are presented. It is also shown that the optimal use of an upper level cache is an NP-hard problem.

Keywords—run-time reconfiguration, multi-level reconfiguration, cache

I. INTRODUCTION

The large amount of context bits that have to be loaded onto modern reconfigurable hardware is a main problem for fast dynamic reconfiguration. One way to speed up dynamic reconfiguration is to adapt the actually available set of reconfigurable resources (and thus the number of reconfiguration bits that are necessary to define the state of the architecture during reconfiguration). Architectures with such a property and which can be dynamically reconfigured on two different levels have been proposed in [6], [7]. On the lower reconfiguration level these architectures allow to perform ordinary reconfiguration operations on the reconfigurable resources. On the upper reconfiguration level the reconfiguration capabilities of the hardware that are available for the lower level reconfigurations can be configured. A high number of reconfigurable resources for example offers high flexibility but slows down dynamic reconfiguration on the lower level because many reconfiguration bits are needed to define the state of all these resources.

Another possibility to obtain fast dynamic reconfiguration is to use a cache for reconfiguration contexts. Then a context that is used several times need not be loaded every time from outside. An example for such architectures are multi context FPGAs. Several strategies for the use of context caches have been considered in the literature. The authors of [9] studied several strategies for caching contexts on FPGAs where the problem is to execute a sequence of calls to a reconfigurable (co)processor. To reduce the total reconfiguration time the authors propose to group subsequences of neighbored calls so that each group of calls fits

into a single context on the FPGA. Different online and off-line algorithms, e.g. based on simulated annealing, for grouping the sequence of calls have been studied in connection with different context replacement strategies for the cache.

Context loading strategies for a reconfigurable architecture where a fixed set of contexts is loaded before a task is executed and other contexts are loaded dynamically was investigated in [13], [14], [17]. Experimental results for the MorphoSys architecture ([12]) which can store 32 contexts show that the proposed strategies lead to reduced configuration latency, power consumption or data and configuration transfer. Other examples of multi context reconfigurable devices that have been introduced in the literature are the dynamically configurable gate array (DPGA) [2], WASMII [10], or the reconfigurable computing module board (RCM) [16]). In order to keep the number of SRAM cells for storing the contexts in multi context FPGAs small it was proposed in [11] to use a decoding stage between context memory and the switches. Another area efficient implementation for a multi context cache that uses redundancy and regularity between the contexts bits has been proposed in [3]. In [21] it was suggested to implement context switching on a reconfigurable device by using look-up tables (LUTs) as shift registers so that each LUT can store one bit of several contexts.

In this paper we propose and study different architectures of context caches for two-level reconfigurable architectures. Since two-level reconfigurable architectures have a context for both levels of reconfiguration (upper and lower level context) it is possible to have a cache for both types of contexts. It has been suggested in [8], [5] that a cache for the upper level contexts might lead to faster dynamic reconfiguration and simple heuristics for the use of the cache have been sketched. So far no experimental results have been given. In this paper we propose refined heuristics and test them extensively with several test applications of different types. We study the influence of the cache size. Moreover, we investigate a problem that allows to change the order of the contexts. This problem is interesting for the design of cache heuristics. It is shown that this problem is NP-hard and heuristics for the problem are presented.

II. 2-LEVEL RECONFIGURABLE ARCHITECTURES

In this section we describe 2-level reconfigurable machines as they have been introduced in [7]. First an intuitive introduction is given before formal definitions are

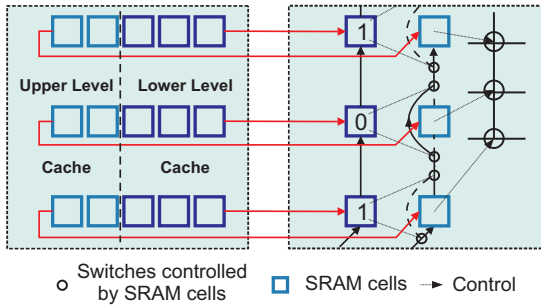


Fig. 1

2-LEVEL RECONFIGURABLE SWITCHBOX (RIGHT BOX) WITH CACHE FOR UPPER AND LOWER LEVEL CONTEXTS (LEFT BOX); EVERY SWITCH HAS TWO CORRESPONDING RECONFIGURATION BITS: THE LOWER LEVEL BIT (LIGHT GREY) DETERMINES WHETHER THE SWITCH IS OPEN OR CLOSED, THE UPPER LEVEL BIT (DARK GREY) DETERMINES WHETHER THE LOWER LEVEL BIT IS INCLUDED IN THE LOWER-LEVEL RECONFIGURATION CHAIN (IF THE UPPER LEVEL BIT IS 1) OR IS SHORTCUT (IF THE UPPER LEVEL BIT IS 0)

presented. A reconfigurable system is called 2-level reconfigurable when the set of resources that are available for ordinary reconfiguration (or the extent to which these resources are available for reconfiguration) is reconfigurable itself. 2-level reconfigurable architectures have two types of reconfiguration steps: i) *ordinary* reconfigurations define the context (in the sense of [4]) of an algorithm and ii) upper level reconfigurations (also called *hyperreconfigurations*) define which reconfiguration capabilities of the reconfigurable resources are available for the subsequent (ordinary) reconfiguration steps (the corresponding context is called upper level context). A central aspect of 2-level reconfigurable architectures is that the reconfiguration costs for lower level reconfigurations (i.e., the number of reconfiguration bits necessary to be loaded onto the architecture) depends on the set of resources that have been made available (by the last upper level reconfiguration). Before we describe the formal model we give an example.

Example: Consider a 2-level reconfigurable switchbox (see the right box in Figure 1). Similar as in an ordinary reconfigurable switchbox, there exists a chain of SRAM cells — called lower level reconfiguration chain — that defines the state of the switches. During a reconfiguration operation the reconfiguration bits are shifted into this register chain. 2-level reconfigurable switch boxes have in addition a second reconfiguration level. For this reconfiguration level there exists a second chain of SRAM cells — the upper level reconfiguration chain — which is used to define which of the SRAM cells of the lower level reconfiguration chain are actually included into the chain. The other SRAM cells are shortcut. During an upper level reconfiguration operation the reconfiguration bits are shifted into this upper level register chain.

We assume that an algorithm/computation can be characterized by a sequence of context requirements that specify for every lower level reconfiguration step which recon-

figurative resources are needed. Each context requirement is a minimal requirement that has to be satisfied in order to guaranty a successful computation. Examples are the number of switches available in the switchboxes in order to satisfy the routing demands or the number of functional units that are available to satisfy the computational demands. Note that the actual lower level reconfiguration operation that is performed might depend on the data and cannot be determined exactly in advance. Therefore the context requirements are upper bounds on the possible lower level reconfiguration demands. When the meaning is clear we call a context requirement of an algorithm/computation sometimes simply a lower level context. In the following we introduce a formal model for 2-level reconfigurable hardware that contains a definition of reconfiguration costs.

Let \mathcal{C} be the set of possible context requirements for a reconfigurable machine (e.g., for a 2-level reconfigurable switch box this is a set of subsets of switches). An algorithm/computation is characterized by a sequence $C = c_1 \dots c_n$ of context requirements $c_i \in \mathcal{C}$, $i \in [1 : n]$ (one context requirement for each reconfiguration operation). An *upper level context* defines the reconfigurable resources which are available in the current state for a lower level reconfiguration. Let \mathcal{H} be the set of possible upper level contexts. For an upper level context $h \in \mathcal{H}$ let $h(\mathcal{C}) \subset \mathcal{C}$ be the set 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 an upper level context h let $c_1 \dots c_k \subset h(\mathcal{C})$ denote the fact that for each context c_i , $i \in [1 : k]$, $c_i \in h(\mathcal{C})$ holds. To bring the hardware into a new upper level context an upper level reconfiguration is performed. For each upper level context $h \in \mathcal{H}$ there exist two costs: i) *init*(h) are the costs to perform an upper level reconfiguration that brings the machine into upper level context h , ii) *cost*(h) are the costs for a lower level reconfiguration step when the machine is in upper level context h .

Thus during the run of an algorithm/computation a sequence of reconfiguration operations $h_1 S_1 \dots h_r S_r$ are performed where h_1, \dots, h_r are upper level reconfigurations and S_i stands for a sequence of lower level reconfigurations which use only those reconfigurable resources that are available within h_i . In the following we describe one model for 2-level reconfigurable hardware that has been discussed for single task applications in [7]. In this model - called switch model - there exists a set of reconfigurable units (called switches) and each subset of this switches can be shortcut from the reconfiguration chain by an upper level reconfiguration (as described before in the switch box example). During a lower level reconfiguration operation the state of each available reconfigurable switch has to be defined. Hence, the cost of a lower level reconfiguration is simply the number of available reconfigurable switches. Formally,

Switch model: Given a set of reconfigurable units or switches $X = \{x_1, \dots, x_n\}$ and a sequence $C = c_1 \dots c_m$ of context requirements with $c_i \in \mathcal{C}$, $i \in [1 : m]$. The set of context requirements \mathcal{C} and the set of upper level contexts \mathcal{H} equal the set of all subsets of X . A switch x is used in

context requirement c if $x \in c$ and otherwise it is unused in c . For switch $x \in X$ 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 and $init(h) = w > 0$ for each $h \in \mathcal{H}$. The total reconfiguration cost of a computation is defined as $cost(C) = r \cdot w + \sum_{i=1}^r |h_i| \cdot |S_i|$, i.e. the sum of all costs for upper level and lower level reconfigurations.

III. CACHES FOR 2-LEVEL RECONFIGURABLE HARDWARE

For 2-level reconfigurable hardware it is possible to use a cache for both reconfiguration levels. We call such a cache a double cache. In the switch model the size of an upper level context and the maximal size of a lower level context are the same. Hence, we propose two types of double caches for the switch model. In a homogenous double cache the size of the storage of a context is the same for upper level and lower level caches. An example of a homogenous double cache for a 2-level reconfigurable switchbox is shown in Figure 1. An advantage of a homogenous double cache is that it is possible to design this cache so that the relative fraction of the cache that can be used for upper level contexts can be changed dynamically. We call this a dynamically partitionable double cache. Thus, in a situation where on a few upper level contexts are needed it is possible to increase the fraction that is used for the lower contexts. The other type of double caches are heterogenous caches where the size of the lower level contexts in the cache can differ. This can increase the total number of contexts that can be stored in the cache.

One important aspect is the time it takes to load a context from the cache. If each part of the cache (i.e. the part for the lower level and the part for the upper level contexts) is realized as a shift-register then the loading time will depend on the size of the corresponding part. Thus in a dynamically partitionable double cache the loading time changes also dynamically.

An interesting aspect of heterogenous double caches is that small lower level contexts might be loaded even when the actually required lower level contexts has additional bits. In this case there are several possibilities how the additional bits are loaded. One possibility is to assume default values for SRAM cells with a content that is not defined by the context that is loaded from the cache. Due to space limitation we have to omit further details on the realization of double caches. In the rest of the paper we concentrate on the use of the partition for the upper level contexts and assume that the costs for loading an upper level context from the cache are zero (because they are very small compared to loading an upper level context from outside).

IV. CACHES FOR UPPER LEVEL CONTEXTS AND THE PERMUTATION PHC PROBLEM

In this section we introduce a problem that is relevant for finding good strategies for the use of an upper level cache for the special case that this cache is very large and that

is also relevant for finding heuristic solutions for loading the upper level cache. We start with an important problem that emerges for a 2-level reconfigurable machine and a given algorithm (i.e. a sequence of context requirements). This problem is to define when upper level reconfigurations have to be done and how corresponding upper level contexts are defined such that the context requirements of the algorithm are satisfied and the total reconfiguration costs are minimized. This problem has been called Partition into Hypercontexts problem (PHC) and it has been shown in [7] that the PHC-Switch problem (i.e. the PHC problem for the switch model) can be solved in polynomial time $O(n \cdot m^2)$ (for a more general model of 2-level reconfigurable architectures it becomes NP-hard). For 2-level reconfigurable machines in the switch model with a cache for upper level contexts it was shown in [8] that the PHC-Switch problem becomes NP-hard (independently which replacement strategy for upper level context is used).

Now we introduce a variant of the PHC problem that is used in the next section. Assume that the upper level context cache can store every possible upper level context, i.e., the upper level context cache has size r_2 with $r_2 \geq |\mathcal{H}|$. Consider an algorithm that is characterized by a sequence of context requirements and assume that the upper level contexts which are used have already been selected. Then it can be observed that the actual sequence of the context requirements has no influence on the total reconfiguration costs. The reason is that every used upper level reconfiguration has to be loaded exactly once and can afterwards be taken from the cache. Since the costs for loading an upper level context from the cache are zero it does not matter when and how often this is the case. Finding the minimal total reconfiguration costs with an infinitely large cache reduces to the problem of finding an optimal assignment of context requirements to upper level contexts without considering the context sequence. We call this problem the Permutation PHC-Switch problem (Perm-PHC-Switch) and can show (the proof is omitted due to limited space):

Theorem 1: The Perm-PHC-Switch problem is NP-complete.

Formally, the Perm-PHC-Switch is defined as follows: Given a 2-level reconfigurable machine in the Switch-model and a sequence $C = c_1 \dots c_m$ of context requirements. Find a permutation π of $1, \dots, m$ and a partition of $C_\pi := c_{\pi(1)}, \dots, c_{\pi(m)}$ into substrings S_1, \dots, S_r (i.e., $C = S_1 \dots S_r$) and upper level contexts h_1, \dots, h_r , $r \geq 1$ with $S_i \subset h_i(\mathcal{C})$ such that the total reconfiguration costs obtained under the PHC-Switch model are minimal.

V. HEURISTICS FOR THE PHC-SWITCH PROBLEM WITH CONTEXT CACHE

In this section we propose heuristic solutions for the PHC-Switch problem with upper level context cache. It can be seen from the proof of Theorem 1 that a solution of the Perm-PHC-Switch problem provides a solution of the PHC-Switch problem with sufficiently large cache. Therefore, we describe in the following two heuristics for the

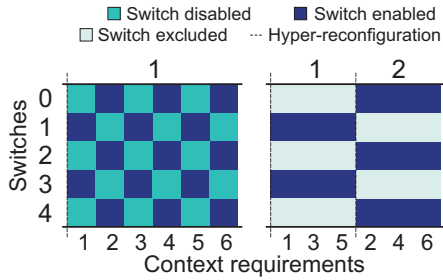


Fig. 2

EXAMPLE OF CONTEXT REQUIREMENTS CONSISTING OF 6 RECONFIGURATION SAMPLES WITH 5 SWITCHES EACH; ORIGINAL SEQUENCE (LEFT), OPTIMAL PERM-PHC-SWITCH SOLUTION (RIGHT)

Perm-PHC-Switch problem and their use for solving the PHC-Switch problem with upper level context caches.

The underlying idea of both heuristics is to sort the context requirements. Therefore the goal is to find an ordering of the context requirements such that the total reconfiguration costs are small. An observation can be made that a reduction of reconfiguration costs is very likely if the resource constraints can be reordered such that individual resources remain unused for a longer time and can thus be removed from the respective upper level contexts. Figure 2 shows such a case.

The first heuristic sorts the context requirements similarly to the Quicksort strategy and is shown in Heuristic 1. The heuristic starts by selecting a pivot switch and then rearranges the context requirements such that those using the selected switch are moved towards the end. The context requirements that do not use the pivot switch are moved towards the beginning of the sequence. This results in two subsequences of context requirements which are then recursively sorted. In every recursion step a different pivot switch is chosen. The selection of the pivot switch is done according to the following steps: (1) Switches which are used in all contexts requirements of the subsequence or are not used in any context requirement are not selected, (2) for each switch x_i (which has not been selected as pivot switch before) the number of switches which are not used in a context requirement if x_i is not used in the same context requirement are added over all context requirements, (3) the switch with the highest sum is chosen as pivot switch.

Switches which are used (respectively not used) in all contexts requirements do not give any information for the sorting of the context requirements and are therefore not chosen as pivot element in the first step. The second step of the algorithm tries to approximate the possible savings that can be obtained when the context requirements where switch x_i is not used are all neighbored. Thus all switches which are also not used in this context requirements are counted because they can possibly contribute to the savings. In the third step the switch with the maximum estimated saving is chosen as pivot.

The second heuristic is based on a strategy similar to insertion sort (see Heuristic 2). For a sequence of con-

Heuristic 1 Quicksort inspired heuristic for the Perm-PHC-Switch problem, parameters (lb,ub)

```

1: if  $lb < ub$  then
2:   for all  $x_i \in X$  do
3:      $sav_i = \max_{j=lb}^{ub} \sum_{x_k \in X} \begin{cases} 1, & \text{if } x_i \notin c_j \wedge x_k \notin c_j; \\ 0, & \text{else} \end{cases}$ 
4:   end for
5:    $pivot\_switch = \arg \max_{i=lb}^{ub} (sav_i)$ 
6:   sort  $c_{lb} \dots c_{ub}$  according to the use of the  $pivot\_switch$ 
7:    $k =$  position of the first context where  $pivot\_switch$  is used (context where it is used come first)
8:   Recurse to  $heuristic1(lb, k - 1)$ 
9:   Recurse to  $heuristic1(k, ub)$ 
10: end if

```

text requirements $S = c_1 \dots c_n$ the sorted sequence S' is generated iteratively. The heuristic starts with the single context requirement with $S'_1 = c_1$. To determine S'_i from S'_{i-1} i candidate sequences are constructed by inserting context requirement c_i at each of the i possible positions in S'_{i-1} . Then the PHC-Switch algorithm is used to calculate $cost(S_{i_j})$ of the j th candidate subsequence S_{i_j} . The candidate subsequence with the least costs is then chosen as S'_i . The algorithm terminates when $S' := S'_n$ was found.

Heuristic 2 Insertion sort inspired heuristic for the Perm-PHC-Switch problem

```

1:  $S'_1 = \{c_1\}$ 
2: for all  $i \in \{2 \dots |S|\}$  do
3:   for all  $pos \in \{1 \dots i + 1\}$  do
4:     Insert  $c_i$  in  $S'_{i-1}$  at position  $pos$  to obtain  $S'_{i_j}$ 
5:     Solve PHC-Switch problem for  $S'_{i_j}$  and calculate  $cost(S_{i_j})$ 
6:   end for
7:    $k = \arg \min_{j=1}^{i+1} cost(S_{i_j})$ 
8:    $S'_i = S'_{i_k}$ 
9: end for
10:  $S_{result} = S'_{|S|}$ 

```

Both heuristics assume that the cache size is large enough to store all upper level contexts and therefore no cache replacements need be performed. However, the cache will usually only be able to store a limited number of upper level contexts at a time. In the following we will describe how both heuristics can be adopted to a limited cache size.

The general idea of Heuristic 3 is to apply heuristic 1 or 2 to prefixes of the sequence of context requirements S . For each prefix the heuristic determines the upper level contexts for all contexts. For a resulting sequence of upper level contexts it is possible to determine the optimal solution for the use of the cache under the assumption that the cache is large enough. Since each upper level context uses the same space in the cache and has the same costs of n for initial loading the longest forward distance (LFD) cache replacement strategy [1] can be used to determine the minimal number of cache misses z . Furthermore the LFD strategy allows to determine the maximal number k' of upper level contexts that are present in the cache at the same time. Then the longest prefix of C can be determined for which this number k' is smaller or equal to the size k of

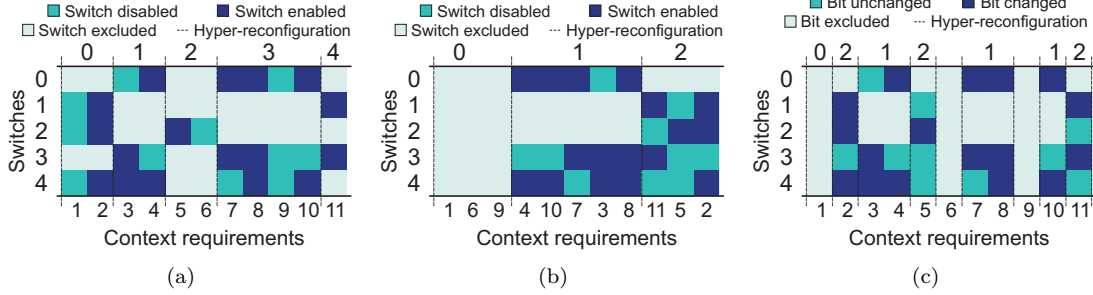


Fig. 3

EXAMPLE OF CONTEXT REQUIREMENTS CONSISTING OF 11 RECONFIGURATION SAMPLES WITH 5 SWITCHES EACH TAKEN FROM THE LUT PART OF AN ALU DESIGN, (A) ORIGINAL SEQUENCE, (B) OPTIMAL PERM-PHC-SWITCH SOLUTION, (C) OPTIMAL CACHE SOLUTION

Heuristic 3 Perm-PHC-Switch problem with limited cache size k , parameters $(heuristic, k)$

- 1: $l = 1, u = k + 1$
 - 2: $j = 1$
 - 3: **while** $u \leq n$ **do**
 - 4: **Call** $heuristic(c_l \dots c_u)$ to compute upper level contexts for $c_l \dots c_u$ /* $heuristic$ is Heuristic 1 or Heuristic 2 */
 - 5: $k' = LFD(\text{upper level contexts for } c_l \dots c_u)$
 - 6: **if** $k' > k$ **then**
 - 7: $S_j = c_l \dots c_{u-1}$
 - 8: $l = u, u = u + k + 1$
 - 9: $j = j + 1$
 - 10: **else**
 - 11: $u = u + k - k' + 1$
 - 12: **end if**
 - 13: **end while**
 - 14: $S_j = c_l \dots c_{|S|}$
 - 15: **Do** postprocessing
-

the cache. The corresponding prefix is removed from the sequence C and the same method is applied iteratively to the resulting sequence until it becomes empty.

In order to make the new heuristic efficient it is not necessary to apply one of the heuristics 1 or 2 to every prefix of C . Instead it is applied first to the prefix of length $k + 1$ because it is clear that a cache of size k is large enough for prefixes of length $\leq k$ prefixes. When the maximal number contexts that are in the cache for this prefix is k' is smaller than k the next prefix that is considered has $k - k' + 1$ contexts more. This is done until a prefix of size l has been found for which $k' > k$. Then the prefix of size $l - 1$ is chosen and removed from the remaining sequence of contexts.

Thus, Heuristic 3 first partitions the sequence S into subsequences S_1, \dots, S_p and computes an upper level context for each context. This is done such that for each subsequence S_i cache size k is enough for the optimal solution that is computed with LFD. Possibly it is necessary to flush the entire cache between two subsequences S_i and S_{i+1} . But this is not always the case. It can happen that an upper level context that is used for a subsequence S_i is already in the cache because it was used for subsequence S_{i-1} . If two contexts h and h' where h is used in S_i and h' is used in S_{i-1} are very similar it might be cheaper to replace them by a context h'' where h'' contains all switches

that are in one of the contexts h or h' . This will increase the costs for lower level configurations but it also saves cost n because h'' has not to be loaded into the cache when S_i is executed. All such possible improvements are checked by Heuristic 3 in a postprocessing step.

Heuristic 2 can be adapted directly to account for a limited upper level cache size and cache misses by introducing additional cache miss costs at line 7. The resulting strategy is presented in Heuristic 4. The cache miss costs comprise of the number of cache misses k times the cache eviction costs of an upper level context n (Note that the $k - 1$ in line 8 stems from the fact that the initial cache miss of each context is already accounted for by the solution of the PHC-Switch problem).

Heuristic 4 Heuristic for Perm-PHC-Switch problem with limited cache size derived from Heuristic 2

- 1: $S'_1 = \{c_1\}$
 - 2: **for all** $i \in \{2 \dots |S|\}$ **do**
 - 3: **for all** $pos \in \{1 \dots i - 1\}$ **do**
 - 4: Insert c_i in S'_{i-1} at position pos to obtain $S'_{i,j}$
 - 5: Solve PHC-Switch problem for $S'_{i,j}$ and calculate $cost(S'_{i,j})$
 - 6: Calculate number of cache misses z using LFD
 - 7: $cost(S'_{i,j}) = cost(S'_{i-1}) + (z - 1) \cdot n$
 - 8: **end for**
 - 9: $k = \arg \min_{j=1}^{i-1} cost(S'_{i,j})$
 - 10: $S'_i = S'_{i,k}$
 - 11: **end for**
 - 12: $S_{result} = S'_{|S|}$
-

VI. EXPERIMENTS AND RESULTS

In this section we present experimental results for the cache heuristics as defined in Section V. The experiments are based on five application problems.

The first application is a 4 bit reconfigurable counter circuit that was implemented on a simple architecture which consists of LUTs as computing elements, a file of registers as storage elements and a MUX and DeMUX as routing elements. For the counter 1 LUT, 1 MUX, 1 DeMUX and 13 registers were used. 48 bits of reconfiguration data are needed to define a context, i.e., the function that is computed by the LUT and the routing that is realized by

TABLE I

TOTAL RECONFIGURATION COSTS (Cost) AND REDUCTION OF TOTAL RECONFIGURATION COSTS COMPARED TO ORDINARY RECONFIGURATION AND NUMBER OF UPPER LEVEL CONTEXTS (# H) FOR THE FIVE TEST APPLICATIONS FOR DIFFERENT MODELS NO RECONFIGURATION: 1-LEVEL RECONFIGURATION (1-LEVEL), 2-LEVEL RECONFIGURATION WITHOUT CACHE (2-LEVEL) WERE THE COSTS OF THE OPTIMAL SOLUTION FOR PHC-SWITCH ARE GIVEN, AND 2-LEVEL RECONFIGURATION WITH CACHE FOR UPPER LEVEL CONTEXTS (2-LEVEL + Cache) WERE THE COSTS OF THE SOLUTIONS COMPUTED BY HEURISTICS 1 AND 2 ARE GIVEN

		1-Level	2-Level	2-Level + Cache	
			PHC-Switch	Heuristic 1	Heuristic 2
FIR	Cost	8000	5078 (63.5%)	3486 (43.6%)	3932 (49.2%)
	# H		64	15	8
Controller	Cost	20304	11024 (54.3%)	7690 (37.9%)	9206 (45.3%)
	# H		20	16	9
VecSum	Cost	496	281 (56.7%)	219 (44.2%)	244 (49.2%)
	# H		5	5	6
Counter	Cost	5280	3761 (71.2%)	2044 (38.7%)	2122 (40.2%)
	# H		30	8	4
LEDDec	Cost	34720	6925 (20.0%)	6985 (20.1%)	6820 (19.6%)
	# H		14	10	11

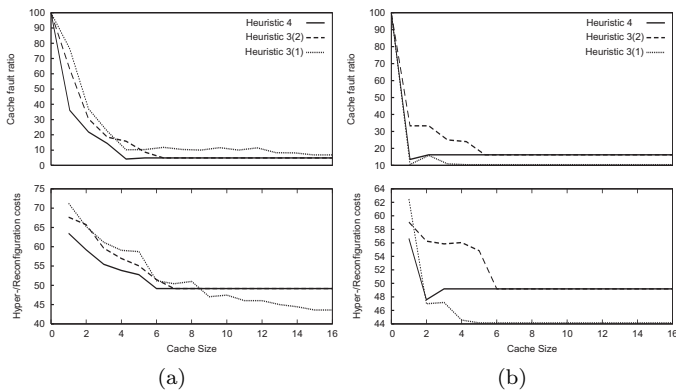


Fig. 4

EXPERIMENTAL RESULTS FOR HEURISTIC 3 USING HEURISTIC 1 (HEURISTIC3(1)) OR USING HEURISTIC 2 (HEURISTIC3(2)) AND HEURISTIC 4 FOR (A) THE VECTOR SUM AND (B) THE FIR FILTER APPLICATION FOR DIFFERENT CACHE SIZES; LOWER ROW: RELATIVE TOTAL RECONFIGURATION COSTS (IN PERCENT OF THE RECONFIGURATION COSTS FOR A 1-LEVEL RECONFIGURABLE ARCHITECTURE); TOP ROW CACHE FAULT RATIO

the MUX and DeMUX. The resulting counter requires 11 reconfiguration operations. A trace of 10 counter cycles are the sample data having a total of 110 reconfigurations. Note, that it is only important here to have the pattern of bit usages which results from the example circuit (more details on how the concrete bit pattern of an algorithm emerges for this architecture can be found in [7]). The second application is a 2-level reconfigurable LED decoder. It was implemented on a similar architecture as the counter but uses 54 registers which leads to 155 contexts of 224 reconfiguration bits each. The third application is a part of a simple controller circuit. Due to the size of the resulting design only the reconfiguration data describing the state of the LUTs were taken into account - one context has 144 bits and 141 reconfiguration operations are done.

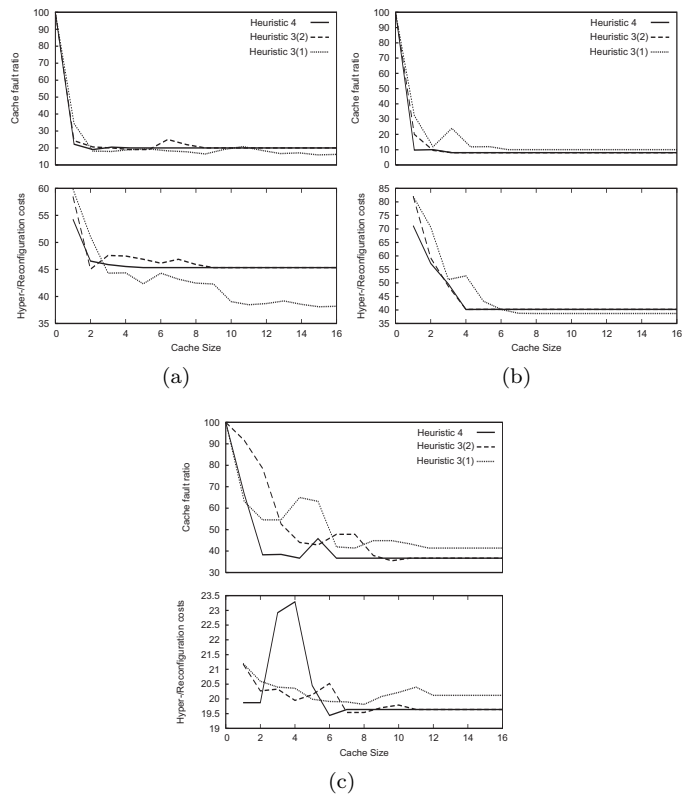


Fig. 5

EXPERIMENTAL RESULTS FOR HEURISTIC 3 USING HEURISTIC 1 (HEURISTIC 3(1)) OR USING HEURISTIC 2 (HEURISTIC 3(2)) AND HEURISTIC 4 FOR (A) THE CONTROLLER, (B) THE COUNTER AND (C) THE LED DECODER APPLICATION FOR DIFFERENT CACHE SIZES; LOWER ROW: RELATIVE TOTAL RECONFIGURATION COSTS (IN PERCENT OF THE RECONFIGURATION COSTS FOR A 1-LEVEL RECONFIGURABLE ARCHITECTURE); TOP ROW CACHE FAULT RATIO

The TMS320C6201 signal processor was used as a coarse granular example. It has 4 types of functional units (L=logic, M=multiply, D=load/store, S=shift) with 2 units for each type. The accompanying C compiler produces optimized and parallelized code and outputs the assembly code, which was used for our simulations. Two example algorithms, a vector summation and an FIR filter were implemented in C, compiled into parallelized VLIW code and executed. During execution the state (idle or busy) of each units was recorded, giving an 8 bit vector for every clock cycle. This sequence of vectors can be seen as a sequence of context requirements for an upper level reconfigurable machine that works as described in connection with the introduction of the PHC-Switch problem. Hence, we used these vectors as input to the PHC-Switch algorithm. The test run with a vector size of 44 bits executes 62 VLIW instructions, (corresponding to 62 context requirements). Likewise an FIR filter was implemented that has 1006 VLIW context requirements.

First we compare the quality of heuristics 1 and 2 for a cache with unlimited capacity. The results in Table I shows that Heuristic 1 works better for most cases than Heuristic 2. Moreover the table shows that for most test applications the upper level context cache could significantly reduce the reconfiguration costs. Only for the LEDDec which uses many different upper level contexts the optimal reconfiguration costs for upper level reconfiguration without cache are smaller than the costs of the solution that was computed with Heuristic 1 when cache is used.

Now we investigate the influence of the upper level cache size on the reconfiguration costs. For all 5 application problems the costs have been computed with heuristics 3 and 4 for different cache sizes (see Figure 4 and Figure 5). Heuristic 3 has been applied in two versions — one version used Heuristic 1 and the other used Heuristic 2. It can be seen in the figures that the total reconfiguration costs are significantly reduced for the 2-level reconfigurable architectures with cache compared to an ordinary 1-level reconfigurable architecture. For all applications the the reconfiguration costs are typically reduced for an increasing cache size. Clearly, for very large cache sizes there is no additional profit. Heuristic 3 when using Heuristic 1 is for most applications the best heuristic when the cache sizes are not too small. For the smaller cache sizes Heuristic 4 performs best for all application architectures.

VII. CONCLUSION

In this paper we have defined cache architectures for 2-level reconfigurable architectures. Moreover, different heuristics for the use of an upper level context cache have been proposed. The heuristics decide when upper level reconfigurations should be done during the run of an algorithm and which upper level contexts should be used in order to reduce the total reconfiguration costs. The heuristics have been evaluated on five example applications which include fine and coarse grain 2-level reconfigurable architectures. It has been shown that for all test application the 2-level reconfigurable architectures can profit from a

second level cache.

REFERENCES

- [1] L. Belady: A Study of Replacement Algorithms for a Virtual-Storage Computer. IBM Systems Journal, 5(2), 78–101, (1966).
- [2] 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).
- [3] W. Chong, S. Ogata, M. Hariyama, M. Kameyama: Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory, 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3, 144-151, (2005).
- [4] A. DeHon, DPGA Utilization and Application. in: Proc. Field-Programmable Gate Arrays (FPGA '96), 115–121, (1996).
- [5] S. Lange, M. Middendorf: Heuristics for Context-Caches in 2-Level Reconfigurable Architectures. Proc. IEEE 2005 Conference on Field-Programmable Technology (FPT' 05), pp. 2, to appear.
- [6] S. Lange, M. Middendorf, Multi Task Hyperreconfigurable Architectures: Models and Reconfiguration Problems. To appear in International Journal of Embedded Systems.
- [7] S. Lange, M. Middendorf, Models and Reconfiguration Problems for Multi Task Hyperreconfigurable Architectures. in: Proc. 11th Reconfigurable Architectures Workshop (RAW 2004), (2004).
- [8] S. Lange, M. Middendorf, The Partition into Hypercontexts Problem for Hyperreconfigurable Architectures. in: Proc. of the International Conference on Field Programmable Logic and Applications (FPL 2004), LNCS 3205, 251-260, (2004).
- [9] Z. Li, K. Compton, and S. Hauck: Configuration Caching for Techniques for FPGA. Proc. IEEE Symposium on FPGAs for Custom Computing Machines, (2000).
- [10] 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).
- [11] A. Lodi, L. Cicarelli, A. Cappelli, F. Campi, M. Toma: Decoder-Based Multi-Context Interconnect Architecture, Proc. IEEE ISVLSI'03, 231-233, (2003).
- [12] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, F. Kurdahi, E.M.C. Filho: "The MorphoSys Parallel Reconfigurable System. Proc. European Conference on Parallel Processing, 727-734, (1999).
- [13] R. Maestre, M. Fernandez, R. Hermida, F.J. Kurdahi, N. Bagherzadeh, H. Singh: Optimal vs. Heuristic Approaches to Context Scheduling for Multi-Context Reconfigurable Architectures, 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, 297-298 (2000).
- [14] R. Maestre, M. Fernandez, F.J. Kurdahi, N. Bagherzadeh, H. Singh: Configuration management in multi-context reconfigurable systems for simultaneous performance and power optimizations, Proc. 13th Int. Symp. System Synth., 107-113, (2000).
- [15] E. Mirsky and A. DeHon, MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources, in: Proc. IEEE Symposium on FPGAs for Custom Computing Machines, 157-166, (1996).
- [16] 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).
- [17] M. Sanchez-Elez, M. Fernandez, R. Hermida, R. Maestre, F.J. Kurdahi, and N. Bagherzadeh: A data scheduler for multi-context reconfigurable architectures. Proc. of the 14th International Symposium on Systems Synthesis, 177-182, (2001).
- [18] S. Sudhir, S. Nath, S. Goldstein: Configuration Caching and Swapping. In Proc. FPL, 192–202, (2001).
- [19] S.M. Scalera and J.J. Murray and S. Lease: A Mathematical Benefit Analysis of Context Switching Reconfigurable Computing. IPPS/SPDP Workshops, 73-78, (1998).
- [20] S.M. Scalera, J.R. Vazquez: The Design and Implementation of a Context Switching FPGA. Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 78-87, (1998).
- [21] J. Torresen. K. A. Vinger: High Performance Computing by Context Switching Reconfigurable Logic. Proc. of 16th European Simulation Multiconference (ESM-2002), 207-210, (2002).