3-2006

Minimizing FPGA Reconfiguration Data at Logic Level

Krishna Raghuraman
Southern Illinois University Carbondale

Haibo Wang
Southern Illinois University Carbondale, haibo@engr.siu.edu

Spyros Tragoudas
Southern Illinois University Carbondale

Published in Raghuraman, K., Wang, H., & Tragoudas, S. (2006). Minimizing FPGA reconfiguration data at logic level. Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06), 224. doi: 10.1109/ISQED.2006.87

Minimizing FPGA Reconfiguration Data at Logic Level

Krishna Raghuraman, Haibo Wang, and Spyros Tragoudas
Southern Illinois University, Carbondale, IL 62901

Abstract

A framework that relates the size of FPGA reconfiguration data to the number of minterms of a specially constructed function is presented. Three techniques, variable mapping optimization, circuit don't-care modification, and look-up table input permutation, are developed to minimize minterms of the special function. The method to integrate the proposed techniques into FPGA design automation flow is discussed and experimental results are presented.

1. Introduction

Reconfigurable systems provide a number of advantages and are continuously gaining their popularity in various applications. Currently, most reconfigurable systems are implemented on FPGA platforms. For such systems, an important design concern is to minimize FPGA reconfiguration bitstreams, and this problem has been widely investigated from high level design. Studies in [1, 2, 3, 4, 5] present different algorithms to perform temporal partitions with the objective of reusing function units in different temporal partitions. Meanwhile, the reuse of FPGA routing patterns is investigated in [6]. Relocation and defragmentation techniques are presented in [7, 8]. The work in [9] minimizes reconfiguration cost by both using coarse-grain logic blocks and optimizing scheduling and allocation schemes. Additionally, other techniques proposed in literature include configuration caching [10], configuration compression [11], and column-based configuration method [12].

Differing from previous approaches, this work addresses the problem of minimizing reconfiguration data at the *logic level*. Techniques developed in this work take advantage of two facts. First, FPGA configuration data are partitioned into frames, which are the smallest data units that can be individually accessed by configuration commands [13]. Second, a frame contains configuration data for identical hardware located in an FPGA column. To conveniently track the size of reconfiguration data, we introduce a framework that links reconfiguration frames to minterms of a specially constructed function, which is referred to as the difference function of a look-up table (LUT) column. Based on this framework, three techniques, variable mapping optimization, circuit don't-care modification, and LUT input order permutation, are proposed to minimize minterms of LUT-column difference functions.

The rest of the paper is organized as follows. Section 2 explains FPGA configuration frames and describes how to link reconfiguration frames to minterms of LUT-column difference functions. Motivational examples are also given in this section to elucidate the proposed techniques. Section 3 develops procedures to efficiently implement the proposed techniques. Section 4 illustrates how to integrate the proposed techniques into FPGA design automation flow and reports experimental results. The paper is concluded in Section 5.

2. Preliminaries

In many LUT-based FPGAs, configuration data are partitioned into frames [13, 14]. A frame contains configuration data for hardware located in an FPGA column. The structure of frames is explained using an FPGA LUT column shown in Figure 1. Assume that there are N LUTs in the column and each LUT has 16 memory locations. The 16 memory locations of any LUT in the column belong to 16 different frames. In addition, each frame contains N bits, corresponding to the same memory locations in the N LUTs of the column. Since a frame is the smallest block of configuration data that can be accessed by configuration commands, the entire frame has to be written into the FPGA even if we just want to change a single bit of an LUT during partial reconfiguration. This arrangement lessens the burden of addressing LUT locations, consequently simplifying hardware design and reducing the size of configuration bitstreams.

As frames are the primitive units of FPGA reconfiguration data, reducing the size of FPGA reconfiguration bitstreams is equivalent to minimizing the number of reconfiguration frames. The latter minimization problem can be addressed in two perspectives. First, it is desirable to have each LUT require less number of frames during reconfiguration. This leads to minimizing the difference between data stored in each LUT before and after reconfiguration. This problem can be tackled by both optimizing variable

Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)
0-7695-2523-7/06 $20.00 © 2006 IEEE

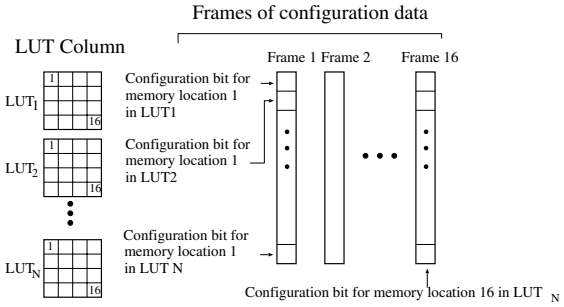


Figure 1. Virtex configuration frames.

mapping and modifying LUT don't-care locations. Before and after reconfiguration, an LUT may implement two different functions that depend on two sets of logic variables. Variable mapping refers to the rule that dictates which two variables (one is an input of the first function and the other is an input of the second function) should be mapped to the same LUT address input. Meanwhile, LUT don't-care locations are memory locations whose addresses correspond to circuit don't-cares. Data stored in don't-care locations can be altered without changing circuit functionality. The second perspective on minimizing reconfiguration frames is to maximize the efficiency of each frame, which is measured by how many bits of the frame containing data that truly update LUT locations. For a given number of LUT locations that need to be updated, higher frame efficiencies will result in less number of frames. The efficiencies of frames can be improved by permuting LUT input orders, which relocates LUT locations that need be updated into common frames.

We first introduce notations used in the paper. We refer to logic functions implemented on an LUT before and after reconfiguration as its *initial* and *final* functions, respectively. For a given LUT, denoted as LUT_i, we use f_i and h_i to represent its initial and final functions. When it is not necessary to distinctively identify LUTs, subscripts of f_i and h_i are omitted for the sake of conciseness. Furthermore, for any given logic function l, we use l^{on}, l^{dc}, l^{off} to represent its on, don't-care, and off sets, respectively.

Three examples will be given to illustrate how variable mapping (*Example 1*), don't-care locations (*Example 2*), and LUT input orders (*Example 3*) can be utilized to reduce reconfiguration frames. Without losing generality, three-input LUTs are used.

Example 1: *For an LUT, assume $f = a \cdot b + c$ and $h = x + y \cdot z$. If the variable mapping is selected as $\{a \leftrightarrow x,\ b \leftrightarrow y,\ c \leftrightarrow z\}$ (symbol $\leftrightarrow$ indicates which two variables are mapped to the same LUT address), two frames (indicated by asterisks) are needed for this LUT as shown in Figure 2. However, if the variable mapping is changed to $\{a \leftrightarrow y,\ b \leftrightarrow z,\ c \leftrightarrow x\}$, no frames are needed.*

Example 2: *For an LUT, assume $f = a \cdot b$ and $h = \overline{a} \cdot \overline{c}$. As shown in Figure 3, four frames are needed for this LUT. However, if both functions have don't-care sets $f^{dc} = \overline{a} \cdot \overline{c} + a \cdot \overline{b}$ and $h^{dc} = a \cdot b + \overline{a} \cdot c$ respectively, then the initial and final functions can be modified as $f^{new} = h^{new} = a \cdot b + \overline{a} \cdot \overline{c}$. No frames are needed after the modification. In this example, both the initial and final functions depend on the same set of logic variables. After the variable mapping is fixed, f and h can have either the same or different support sets.*

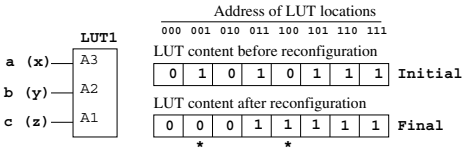


Figure 2. LUT data without variable mapping optimization.

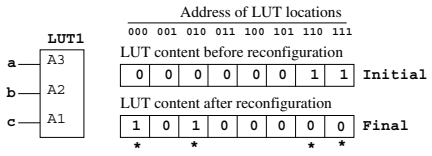
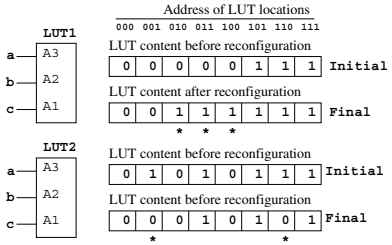
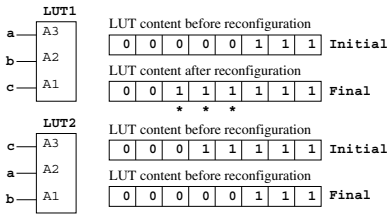


Figure 3. LUT data without don't-care modification.

Example 3: *Assume LUT_1 and LUT_2 are in the same column and $f_1 = a \cdot (b + c)$, $h_1 = a + b$, $f_2 = a \cdot b + c$, $h_2 = (a + b) \cdot c$. If the input orders for both LUTs are $\{a \rightarrow A_3,\ b \rightarrow A_2,\ c \rightarrow A_1\}$, five frames are needed as shown in Figure 4(a). However, if the input order for LUT_2 is changed to $\{c \rightarrow A_3,\ a \rightarrow A_2,\ b \rightarrow A_1\}$, only three frames are required as shown in Figure 4(b). Note that LUT input order permutation is performed with fixed variable mappings. During the permutation , LUT input orders for both initial and final functions are changed in the same way.*



(a) Before LUT input permutation.



(b) After LUT input permutation.

Figure 4. LUT data with input permutation.

Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)
0-7695-2523-7/06 $20.00 © 2006 IEEE

In the quest for solutions of the proposed minimization problem, we are more interested in how logic values (0 or 1) are stored in LUTs, rather than what actual functions implemented on LUTs are. Due to this reason, we introduce the concept of the *LUT mapping function*. LUT mapping functions are LUT functions expressed in terms of LUT address variables. For an LUT whose implemented logic function is given, we can obtain its mapping function through substituting logic variables by their associated address variables. For example, the initial function of LUT_1 in Figure 4 is $a \cdot (b + c)$. Substituting logic variables by their associated LUT address variables, we have its mapping function as $A_3 \cdot (A_2 + A_1)$. The mapping function of an LUT represents all the LUT locations that store logic 1. Since each LUT is associated with two logic functions (f and h), there are two mapping functions for each LUT as well. Due to the close relation between LUT logic functions and their corresponding mapping functions, we also use f and h to represent the initial and final mapping functions of an LUT, respectively.

Based on LUT mapping functions, we define the LUT difference function as:

$$D = f \oplus h \tag{1}$$

In addition, the difference function of an LUT column is defined as:

$$\mathcal{D} = \bigcup_{i=1}^{N} D_i \tag{2}$$

where, N is the total number of LUTs in the given column and D_i is the LUT difference function of LUT_i. In the computation of $\mathcal{D}$, address variables with the same name but located in different LUTs (e.g. A_1 of LUT_i and LUT_j) are treated as the same variable, since they function as coordinates to indicate LUT locations containing logic 1. Therefore, function $\mathcal{D}$ depends on only p variables: A_p, A_{p-1}, $\cdots$, A_1, where p is the number of inputs of the LUTs in the column. It is easy to see that the number of minterms in $\mathcal{D}$ is equal to the number of frames requested for reconfiguring the entire LUT column. Due to this reason, the phrase of minimizing LUT difference functions is used in the rest of the paper as a convenient synonym of minimizing the number of minterms in LUT difference functions.

3. Proposed Techniques

As discussed early, FPGA reconfiguration data can be minimized by optimizing variable mapping, modifying LUT don't-care locations, and permuting LUT input orders. The problem of finding optimal variable mappings is easy since it can be solved separately for each LUT. Techniques to perform the other two optimization procedures are discussed in the following.

3.1 Modifying LUT don't-care locations

In general, *expressions* for f and h of an LUT contain their entire on sets (f^{on} and h^{on}) and portions of their don't-care sets (f^{dc} and h^{dc}). We use $f^{dc\star}$ and $f^{dc\dagger}$ to distinguish don't-cares of f that are *included* and *excluded* in the expression of f. Similar notations apply to function h. Then, we have $f = f^{on} + f^{dc\star}$ and $h = h^{on} + h^{dc\star}$. The LUT difference function can be written as:

$$\begin{aligned} D &= f \cdot \overline{h} + \overline{f} \cdot h \\ &= f^{on} \cdot h^{off} + f^{on} \cdot h^{dc\dagger} + f^{dc\star} \cdot \overline{h} \\ &+ h^{on} \cdot f^{off} + h^{on} \cdot f^{dc\dagger} + h^{dc\star} \cdot \overline{f} \end{aligned} \tag{3}$$

Obviously, $f^{on} \cdot h^{off} + h^{on} \cdot f^{off}$ constitutes the lower bound of the difference between f and h. The other terms on the right-hand-side of Equation 3 can be eliminated by assigning proper values to LUT don't-care locations. This is formally stated by the following corollary.

Corollary 1 *The number of minterms of an LUT difference function is minimized if the initial and final functions of the LUT are modified as follows:*

$$f^{new} = f + f^{dc} \cdot h - f^{dc} \cdot \overline{h} - f^{dc} \cdot h^{dc} \tag{4}$$

$$h^{new} = h + h^{dc} \cdot f - h^{dc} \cdot \overline{f} - f^{dc} \cdot h^{dc} \tag{5}$$

In the above equations, symbols $+$, $\cdot$, and $-$ represent set union, intersection, and subtraction operations. For an LUT, adding a minterm to its function implies changing the value stored in the LUT location that corresponds to the minterm to logic 1. Meanwhile, subtracting a minterm is the same as putting logic 0 to the corresponding LUT location. It is easy to show $f^{new} \oplus h^{new} = f^{on} \cdot h^{off} + h^{on} \cdot f^{off}$ and, hence, prove the corollary. By performing function modification according to the above corollary, minterms added to f are:

$$\mu^{+} = f^{dc} \cdot h - f^{dc} \cdot h^{dc} - f \tag{6}$$

Similarly, minterms that are subtracted from f can be expressed as:

$$\mu^{-} = f^{dc} \cdot \overline{h} + f^{dc} \cdot h^{dc} - \overline{f} \tag{7}$$

The total LUT locations that are altered can be expressed by their corresponding minterms as:

$$\mu = \mu^{+} + \mu^{-} \tag{8}$$

Note that a similar set of equations apply to function h.

For an LUT, its don't-cares consist of controllability don't-cares (CDCs) and observability don't-cares (ODCs). CDCs are signal patterns that never appear at the LUT inputs. Meanwhile, ODCs are defined as LUT input patterns

representing scenarios that the LUT output cannot be observed by circuit primary outputs. Because CDC sets of different LUTs are independent of each other, modifying LUT locations addressed by CDC patterns can be performed individually for each LUT. This simple process always leads to the globally optimized solution when only CDCs are under consideration. On the contrary, modifying ODC locations is a complicated process. When ODC locations of an LUT are modified, ODCs of other LUTs may change. Although it is theoretically possible to re-compute ODCs for the rest of LUTs after each LUT is modified, this approach is practically unattractive due to its computation complexity. To avoid repeated re-computation of LUT ODCs, this section presents an efficient method to compute LUT ODCs that can be simultaneously modified, which are referred to as compatible ODCs (CODCs). To address a similar problem in logic synthesis, several techniques [15, 16, 17, 18] have been proposed. The method presented here is similar to approaches discussed in [16, 17] in the perspective of computing CODC upper bounds. However, it differs from the previous approaches in the following two aspects. First, ODCs covered by their upper bounds are further restricted according to Equation 8. Second, a heuristic method is utilized to determine the order of LUTs to be processed.

The simultaneous optimization for multiple vertices (gates or LUTs), denoted as $y_1, y_2, \cdots, y_n$, can be modeled by n perturbation variables $\delta_1, \delta_2, \cdots, \delta_n$ [15]. In this application, δ_i represents ODCs that are added or subtracted from the function of LUT_i. Let $\mathbf{DC}^{ext}$ represent external don't-cares, $\mathbf{ODC}^{y_i}$ denote ODCs at vertex y_i, and symbol $|$ represent generalized cofactor operations. A sufficient condition for the equivalence between the perturbed and original circuits is [16]:

$$\delta_i \mathbf{1} \subseteq \mathbf{DC}^{ext} + \mathbf{ODC}^{y_i}|_{\delta'_1,\cdots,\delta'_{i-1}} \quad i = 1, 2, \cdots n. \quad (9)$$

In the above expression, don't-cares with respect to different primary outputs are represented in the vector format and $\mathbf{1} = (1, 1, \cdots, 1)$. The above condition gives a series of upper bounds (with respect to different primary outputs) for δ_i, which depend on $\mathbf{ODC}^{y_i}$ and previous perturbations. Let m denote the number of circuit primary outputs, DC_j^{ext} and $ODC_j^{y_i}$ denote the external and observability don't-care sets at vertex y_i with respect to primary output j, respectively. The global upper bound, which is in the scalar format, can be obtained as:

$$\zeta_i(\delta_1, \cdots, \delta_{i-1}) = \bigcap_{j=1}^{m} (DC_j^{ext} + ODC_j^{y_i}|_{\delta'_1,\cdots,\delta'_{i-1}}) \quad (10)$$
$$for\ i = 1, 2, \cdots n$$

As FPGA reconfiguration data for an FPGA column depend on all the LUT functions of the column, it is imperative to simultaneously optimize all LUT functions of a column. In addition, LUT difference functions with large numbers of minterms are likely to affect the overall reconfiguration frames. Therefore, such LUTs should be given high priorities during the optimization. Due to this observation, the proposed procedure first ranks all the LUTs according to the number of minterms in their difference functions. LUTs whose difference functions contain more minterms are given higher ranks. Following the descending order of LUT ranks, ODCs are pruned in accordance with two constraints. The first constraint is Equation 8, which eliminates ODCs that don't minimize LUT difference functions. The second constraint is the upper bound given in Equation 10, which is used to guarantee the correctness of the resulted circuit.

The proposed procedure is further elaborated as follows. For the convenience of description, we re-label LUTs after ranking such that LUTs with higher ranks are given smaller index numbers. For example, N LUTs arranged in the descending order of their ranks will be listed with their new labels as $LUT_1, LUT_2, \cdots, LUT_N$. Thus, LUT_1 is the first LUT to be processed. When the initial function of LUT_1 is under consideration, LUT locations whose values are desired to be altered are:

$$\delta_1^f = \mu_1^f \quad (11)$$

In the above and following equations, we use superscripts to indicate the function on which δ and μ are defined. Also, we use subscripts to indicate the LUT that δ and μ are associated with. Since LUT_1 is the first LUT to be processed, δ_1^f is not subject to the second constraint. However, when LUT_k ($k \neq 1$) is processed, we have to apply both constraints. This leads to:

$$\delta_k^f = \mu_k^f \cdot \zeta_k^f(\delta_1, \cdots \delta_{k-1}) \quad (12)$$

The pseudo-code of the proposed CODC computation procedure is given in Figure 5. Note that CODCs for both LUT initial and final functions are computed simultaneously in the procedure.

```
ODC_OPT( LUTs ) {
1   Compute ODCs for all LUTs regarding
    their initial and final functions
2   Rank all LUTs and re-label them according
    to the descending order of their ranking
3   δ_1^f = μ_1^f;  δ_1^h = μ_1^h
4   for k=2 to N
5       δ_k^f = μ_k^f · ζ_k^f(δ_1^f, ··· δ_{k-1}^f)
6       δ_k^h = μ_k^h · ζ_k^h(δ_1^h, ··· δ_{k-1}^h) }
```

Figure 5. CODC computation procedure.

3.2 Permuting LUT input orders

By defining LUT-column difference function $\mathcal{D}$, we relate the number of reconfiguration frames to the number of

minterms in $\mathcal{D}$. Thus, the optimal LUT input orders should minimize minterms in the corresponding column difference function. Although it is possible to solve this problem through exhaustive enumeration, the large search space of this problem makes a such approach impractical. This paper presents a search procedure based on a greedy algorithm. With assumptions that each LUT has p inputs and N LUTs are in the give column, the major steps of the procedure are described in Figure 6. It first constructs LUT difference functions (line 3) and, concurrently, finds the LUT that requires the least number of reconfiguration frames (lines $4 \sim 5$). The input order of that LUT will not be permuted, and is used as a reference when permuting other LUT input orders. Also, function *MintermCount* used in line 3 counts the number of minterms of its operand. After the reference LUT is selected, the algorithm sequentially picks an unprocessed LUT and permutes its inputs. The permutation procedure is sketched from lines 9 to 18. It exhaustively tries all the possible permutations and picks the one that results in the smallest increase on the number of minterms of the newly constructed union function ($\mathcal{D}^{tmp}$). The time complexity of the proposed procedure is $(p!) \cdot (N-1)$, which is significantly smaller than the time complexity of the exhaustive enumeration method.

```
1   min_tmp = 2^p
2   for i = 1 to N
3     D[i] = f_i ⊕ h_i;  min = MintermCount(D[i])
4     if min < min_tmp
5       min_tmp = min;  min_index = i;  D = D[i]
6   for i = 1 to N
7     if i ≠ min_index
8       D = permute(D, D[i])

9   permute( D, D[i] ) {
10    min_tmp = 2^p
11    for each permutation order of LUT_i
12      derive new function D'[i] according
          to the new input order
13      D^tmp = D ⋃ D'[i]
14      min = MintermCount( D^tmp )
15      if min < min_tmp)
16        min_tmp = min;  D^min = D^tmp
17        Order[LUT_i] = current permut. order
18    return D^min }
```

Figure 6. LUT input permutation procedure.

4. Experimental Results

This section describes how the proposed techniques can be integrated into FPGA design automation flow, and reports experimental results. The current FPGA design automation flow is sketched by the solid arrows in Figure 7(a). For reconfiguration applications, FPGA implementations of both initial and final circuits are generated following the same flow. The reconfiguration bitstreams, which change FPGA hardware from the initial circuit to its final circuit, are produced by comparing the initial and final FPGA implementations. The proposed optimization procedures can be added into the design flow between placement and routing steps as shown in Figure 7(b). After the placement phases of both the initial and final circuits, the initial and final functions of all the LUTs become available. Hence, the proposed techniques can be applied to optimize variable mappings, modify LUT don't-care locations, and find optimal LUT input orders. After this, FPGA routing can be performed accordingly.

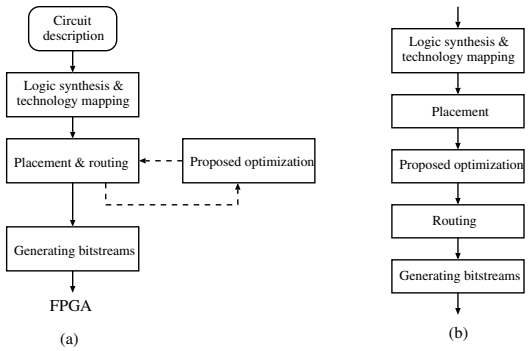


Figure 7. Integrating the proposed techniques into FPGA design flow.

It is often difficult to have direct access to results produced by the FPGA placement procedure. In this case, our method can be integrated as indicated by the dash arrows in Figure 7(a). After the placement and routing (*P&R*) phases of both the initial and final circuits, we let the FPGA tool write *P&R* results into structural VHDL files. The basic components in these VHDL files are LUTs. In addition, we let the FPGA tool generate location constraints for each LUT in VHDL files according to *P&R* results. The VHDL files along with the constraint files provide information about LUTs in the same column and their initial and final functions. After applying the proposed optimization procedures, LUT $init$ values (that represent LUT locations storing logic 1) are updated and new constraints regarding LUT input orders are added into constraint files. The updated VHDL and constraint files are fed to the *P&R* module in the FPGA tool to re-route FPGA circuits.

We experimented with the latter integration scenario. Due to the lack of suitable partial reconfiguration benchmark circuits, we use ISCAS85 benchmark circuits as initial FPGA circuits. We derive final FPGA circuits by performing random function modification on the initial circuits. In this process, we first define a set of functions, denoted as $g_1, g_2, \cdots g_i$, which depend on variables $A_4,\ A_3,\ A_2,\ A_1$ (since four-input LUTs are used in our experiments). Then, we derive final LUT functions by performing either *COMPOSE* or *INTERSECT* operation with using the original LUT function and one function selected from $g_1, g_2, \cdots g_i$

as operands. The *COMPOSE* and *INTERSECT* are function manipulation operations defined in CUDD package that is used in the implementation of our optimization procedures. The selection on operation (*COMPOSE* or *INTERSECT*) and operand function ($g_1, g_2, \cdots g_i$) is totally randomized.

The experiments are conducted on Xilinx Virtex 1000 platform. The obtained results are summarized in Table 1. The second column of the table lists the number of LUTs assigned to each column. Several column configurations are investigated in the experiment. The third column records the required frame numbers without performing any of the proposed optimization. The fourth column summarizes the number of frames contained in reconfiguration data when only LUT input order permutation technique is applied. The percentage of frame reduction is given in the fifth column. With both don't-care modification and LUT input order permutation techniques being utilized, the resultant reconfiguration frame numbers and their corresponding saving (in percentage) are summarized in the sixth and seventh columns, respectively. The results show that the proposed techniques can reduce reconfiguration frames by more than 20% on average.

Table 1. Comparing Reconfiguration frames.

Circuit	#lut	W/o. Opt.	Inp. Perm. Only		DC Opt. & Inp. Perm.	
			#Frm.	R(%)	#Frm.	R(%)
C432	3	274	244	11%	236	14%
	4	238	212	11%	208	13%
	8	166	136	18%	136	18%
C1355	3	142	137	4%	117	18%
	6	124	111	10%	99	20%
	9	106	95	10%	83	22%
C1908	3	255	239	6%	143	44%
	6	198	175	12%	119	40%
	9	172	141	18%	91	47%
C2670	3	430	389	10%	322	25%
	6	334	286	14%	251	25%
	9	276	232	16%	204	26%
C3540	6	771	659	15%	580	25%
	9	632	506	20%	452	28%
	12	567	409	28%	377	34%
C5315	9	769	617	20%	529	31%
	12	626	574	8%	505	19%
	15	542	440	19%	402	26%
C6288	12	1168	964	17%	849	27%
	15	986	826	16%	786	20%
	18	852	712	16%	672	21%
C7552	12	967	780	19%	686	29%
	15	814	660	19%	611	25%
	18	693	570	18%	539	22%

5. Concluding Remarks

This paper presents a comprehensive methodology to minimize FPGA reconfiguration data at logic level. The methodology is based on a framework that links the size of reconfiguration data to the number of minterms contained in LUT-column difference functions. It comprises three techniques, which are variable mapping optimization, don't-care location modification, and LUT input order permutation. To efficiently implement the proposed techniques, two heuristic algorithms are developed for computing compatible don't-care locations and finding optimal LUT input orders from a large search space. The developed techniques can be perfectly combined with other methods that minimize FPGA reconfiguration data at high levels for further reducing FPGA reconfiguration cost.

References

[1] J. M. Cardoso, "On Combining Temporal Partitioning and Sharing of Function Units in Compilation for Reconfigurable Architectures," *IEEE Trans. on Computers*, vol. 52, no. 10, pp. 1362–1375, 2003.

[2] M. Meribout and M. Motomura, "Efficient Metrics and High-Level Synthesis for Dynamically Reconfigurable Logic," *IEEE Trans. on VLSI*, vol. 12, no. 6, 2004.

[3] M. Kaul and R. Vemuri, "Temporal Partitioning Combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs," in *Proc. DATE*, pp. 202–209, 1999.

[4] M. Kaul and R. Vemuri, "An Automated Temporal Partitioning and Loop Fission Approach for FPGA Based Reconfigurable Synthesis of DSP Applications," in *Proc. DAC*, pp. 616–622, 1999.

[5] K. M. GajjalaPurna and D. Bhatia, "Partitioning in time: a paradigm for reconfigurable computing," in *Proc. ICCD*, pp. 340–345, 1998.

[6] D. Rakhmatov and S. B.K. Vrudhula, "Minimizing routing configuration cost in dynamically reconfigurable FPGAs," in *Proc. Parallel and Distributed Processing Symp.*, pp. 1481–1488, 2001.

[7] K.Compton, J.Cooley and S.Knol, "Configuration relocation and defragmentation for reconfigurable computing," in *Proc. IEEE Symp. FPGA Custom Computing Machines*, pp. 79–80, 2000.

[8] K.Compton, Z.Li,S.Knol and S.Hauck, "Configuration relocation and defragmentation for reconfigurable computing," *IEEE Trans. on VLSI*, vol. 10, pp. 209–220, 2002.

[9] Z. Huang and S. Malik, "Managing dynamic reconfiguration overhead in SoC design using reconfigurable datapaths and optimized interconnect networks," in *Proc. DATE*, pp. 13–16, 2001.

[10] Z. Li, K. Compton, and S. Hauck, "Configuration Caching for FPGAs," in *Proc. IEEE Symp. FPGA Custom Computing Machines*, pp. 22–36, 2000.

[11] S. Hauck, Z. Li, and E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA," in *Proc. FPGA Custom Computing Machines*, 1998.

[12] S. Mitra, W. Huang, N. Saxena, S. Yu, and E. J. McCluskey, "Reconfigurable Architecture for Autonomous Self-Repair," *IEEE Design and Test of Computer*, vol. 21, no. 2, pp. 228–240, 2004.

[13] XILINX Inc., *Virtex Series Configuration Architecture User Guide*, 2003.

[14] XILINX Inc., *Two Flows for Partial Reconfiguration:Module Based or Small Bit Manipulations*, 2002.

[15] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[16] M. Damiani and G. De Micheli, "Don't Care set Specifications in Combinational and Synchronous Logic Circuits," *IEEE Trans. on CAD*, vol. 12, no. 3, pp. 365–388, 1993.

[17] H. Savoj and R. Brayton, "The use of Observability and External Don't cares for the Simplification of Multi-Level Netwworks," in *Proc. DAC*, pp. 297–301, 1990.

[18] S. Yamashita, H. Sawada, and A. Nagoya, "SPFD: A New Method to Express Functional Flexibility," *IEEE Trans. on CAD*, vol. 19, no. 8, pp. 840–849, 2000.

Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)
0-7695-2523-7/06 $20.00 © 2006 IEEE