Power and Delay Reduction via Simultaneous Logic and Placement Optimization in FPGAs

Balakrishna Kumthekar and Fabio Somenzi
Dept. of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309

Abstract

Traditional FPGA design flows have treated logic synthesis and physical design as separate steps. With the recent advances in technology, the lack of information on the physical implementation during logic synthesis has caused mismatches between the final circuit characteristics (delay, power and area) and those predicted by logic synthesis. In this paper, we present a technique that tightly links the logic and physical domains—we combine logic and placement optimization in a single step. The combined algorithm is based on simulated annealing and hence, very amenable to new optimization goals or constraints. Two types of moves, directed towards global reduction in the cost function (linear congestion), are accepted by the simulated annealing algorithm: (1) logic optimization steps consisting of removing or replacing redundant wires in a circuit using functional flexibilities derived from SPFDs [12] and (2) the placement optimization steps consisting of swapping a pair of blocks in the FPGA. Feedback from placement is very valuable in making an informed choice of a target wire during logic optimization moves. Experimental results demonstrate the efficacy of our approach over the placement independent approach.

1. Introduction

FPGAs consist of a large number of programmable logic blocks, which can implement a small amount of digital logic, and programmable routing which allows the inputs and outputs of logic blocks to be connected to form larger circuits. In a Look Up Table (LUT) architecture, each logic block can implement any *k*-input Boolean function.

The synthesis flow for FPGAs typically consists of four steps. In the first step, a logic synthesis tool [8] performs technology independent logic optimization to transform the design into a multi-level network. In the second step, a technology mapper [8, 4] maps the circuit onto logic blocks to minimize the number of lookup tables and/or the depth of the circuit. Placement and routing form the final two steps. Traditional placement and routing tools for FPGAs accomplish the job based solely on the connection structure of the multi-level network. This practice more often than not leads to congestion of wiring channels in certain areas of FPGA, resulting in longer wiring delay and increased power dissipation. There is no systematic way to make incremental changes to satisfy design constraints. Such changes, however, are unpredictable and may amount to unproductive design iterations.

Most circuit delay in FPGAs is due to routing delays, rather than logic block delays, and most of an FPGA area is devoted to routing. Hence, logic techniques that perform *intelligent* wire removal and/or replacement are essential to improve circuit characteristics and reduce design iterations. In this paper, we present a technique to perform simultaneous logic and placement optimization to reduce delay and power dissipation in circuits mapped to FPGAs. The combined algorithm is based on simulated annealing and hence, very amenable to new optimization goals or constraints. Two types of moves, directed towards global reduction in congestion, are performed by the simulated annealing algorithm: (1) the logic optimization steps consisting of removing or replacing redundant wires in a circuit using functional flexibilities derived from SPFDs [12] and (2) the placement optimization steps consisting of swapping of a pair of blocks in the FPGA. Feedback from placement is very valuable in making an informed choice of a target wire during logic optimization moves. Logic optimization steps that do not have such a feedback, remove or replace wires with little or no beneficial effect on the final placed and routed circuit.

In previous work on FPGAs, the authors of [3] present a post-layout optimization technique to perform wire replacement using implication based techniques. Wires that have alternate wires for replacement are identified during logic optimization and are used during routing. This information is used to replace an unroutable wire by a poten-

tially routable wire chosen from the replacement set. The authors of [9], working independently, presented a combined logic and placement optimization technique for standard cell based designs. To the best of our knowledge, our method is the first to combine logic and placement optimization for FPGAs. The technique presented in this paper can also be incorporated in the framework of [9].

The rest of this paper is organized as follows. In Section 2 we briefly describe SPFDs and present an algorithm to remove/replace wires at the technology independent level. This section also presents results and discusses the shortcomings of this approach. Simultaneous placement and logic optimization algorithm and its benefits are presented in Section 3. Finally, conclusions are presented in Section 4.

2. Placement Independent Logic Optimization

The authors of [12], introduced a new approach, SPFDs, for expressing the flexibility that a node can have in a multi-level network. The don't cares represented by SPFDs can be seen as a set of incompletely specified functions. They can be computed efficiently and are independent, like compatible sets of permissible functions (CSPFs) [7], but express more flexibility; SPFDs are a generalization of CSPFs. SPFDs can be exploited to perform logic optimization by removing/replacing wires and nodes in a network. For more details the reader can refer to [12, 10].

We have applied SPFDs (based on the idea of cluster of CLBs [6]) to remove/replace redundant wires in the circuit. SPFDs can be computed for the entire network starting from the primary outputs in reverse topological order. However, this is feasible only for small networks. In order to handle large circuits we constrain ourselves to nodes in smaller regions (clusters) of the circuit and compute SPFDs only for such nodes. Working with clusters reduces not only the amount of time required to optimize a circuit but also the memory requirements.

Figure 1 illustrates the concept of clusters. For a given target wire, we compute the *fanout cluster* up to a user specified depth. The depth can be anywhere between 1 (immediate fanout) and the maximum topological depth of the network. A depth-first search, from the target wire up to the user specified depth, is performed to compute the cluster nodes. Nodes in the cluster with highest topological depth are called *boundary* nodes. The nodes within the cluster are called *internal* nodes. The SPFDs for boundary nodes are computed from their local ON-set and OFF-set. This ensures that any changes made within the cluster need not be propagated beyond the boundary nodes. This setup drastically reduces the time to perform costly fanout function updates. SPFDs can be computed for every node in the cluster by starting at the boundary nodes [12]. The order in which the edges of node SPFD are distributed to its fanin is chosen according to the heuristics outlined in Section 2.1.

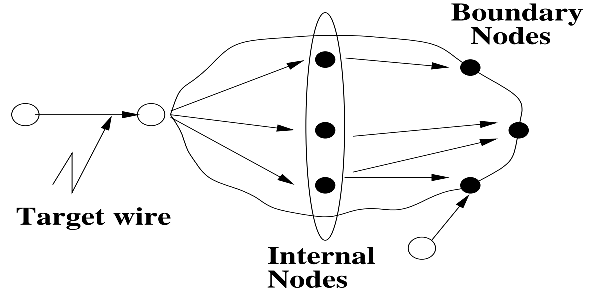

Figure 1. Fanout cluster.

2.1. Heuristics

We have implemented many heuristics and incorporated memory optimizations, to accomplish the wire removal and wire replacement task. The following heuristics are employed to select a node whose fanout wires we target for removal or replacement.

- **Switched capacitance**: When the optimization objective is power reduction, nodes in the circuit are sorted according to their switched capacitance, SWC. The node switching activity is computed via simulation of representative set of user specified patterns or randomly generated patterns based on user specified primary input statistics. During placement-independent wire removal or replacement, the fanout count of the node is used as an estimate of load capacitance. When this heuristic is applied during simultaneous logic and placement optimization in Section 3, the half-perimeter length of the net bounding box, weighted by the fanout count, is used as an estimate of the load capacitance. Also, during SPFD propagation, the switched capacitance is used to order fanin nodes. During optimization, when an alternative function is extracted from the node SPFD, the one that reduces node switching activity the most is chosen.

- **Topological depth**: As in the switched capacitance heuristic, the topological depth of a node is used to sort the fanin nodes during SPFD propagation. A node closer to the primary inputs is set to have lower priority (less flexible) than a node farther away. The rationale behind this choice is that a loosely constrained (more flexibility) node can propagate more flexibility information to its fanin than a tightly constrained node. A convex combination, $\alpha \cdot SWC + (1-\alpha) \cdot depth$, $0 \leq \alpha \leq 1.0$, of switched capacitance and topological depth can also be used.

- **Depth of cluster**: The *depth* of the cluster is a user-defined option. The depth of the cluster can be varied to trade-off memory and computational time requirements against functional flexibility extracted. The same depth is used to limit the search for potential nodes whose output can replace the source node of the current target wire.

Memory optimization is a primary concern in any software implementation. In the implementation of the algorithm, efficient techniques were used to ensure reuse of BDD variables. Auxiliary variables are required during the computation of SPFDs. Instead of allocating new BDD variables, necessary auxiliary variables are chosen from the pool of existing BDD variables. The choice of a variable is made in such a way that the resulting Boolean functions do not result in memory blow-up. For example, when computing a Boolean relation, BDD variables (representing the individual functions) which are at the top of the BDD variable order are chosen. In all the experiments we ran, memory explosion was never a problem even for the largest benchmarks. Memory usage seldom exceeded 50MB.

2.2. Basic Algorithm

```
for (iter = 0; ; iter++) {
  if(useSWC)
    if(iter % N == 0)
      PerformCompleteSimulation();
    else
      PerformPartialSimulation();
  node = PickUnlockedNode(network, SortHeuristic);
  if (node == NULL) {
    break;
  } else {
    ProcessFanoutWires(node, maxDepth);
    LockNode(node);
  }
}

Core:
ProcessFanoutWires(node, maxDepth) {
  foreach(targetWire ∈ fanoutWires(node)) {
    cluster = SelectCluster(targetWire, maxDepth);
    ComputeClusterSPFDs(cluster);
    ComputeWireSPFD(targetWire);
    if (IsWireRemoveable(targetWire)) {
      RemoveWire(targetWire);
    } else if (IsWireReplaceable(targetWire)) {
      ReplaceWire(targetWire);
    }
  }
}
```

Figure 2. SPFD-based wire removal and replacement.

Figure 2 provides an outline of the placement-independent wire removal and replacement algorithm. If the switched capacitance heuristic is used, pattern based simulation is performed to compute the switching activity for the nodes in the circuit. Complete simulation, using all the specified vectors, is performed only at the beginning of the algorithm. Every N iterations, partial simulation (using only a portion of the specified vectors) is performed to get a rough estimate of switching activity. Initially all the nodes in the network are *unlocked*, i.e., all the nodes are available for optimization. As optimization progresses, every node whose fanout wires have been examined are locked and hence, are not selected in future iterations. The algorithm ends when all the nodes are locked.

Nodes in the network are ordered according to their switched capacitance. For every *node* that is selected, the fanout *cluster* is computed according to a specified *maxDepth*. SPFDs are then computed for the cluster nodes and the *targetWire*. If the SPFD of the target wire is empty then it is removed, else if a node exists whose function satisfies the wire SPFD, the target wire is replaced. The node is then locked after examining all the fanout wires.

2.3. Results

We have implemented the above algorithm in VIS [2]. We use CUDD [11] as the underlying BDD package. We have tested our algorithm on several MCNC benchmark circuits. All original circuits were fed to SIS [8] for logic optimization and mapping onto 4-input LUTs. The script we used for this task is the one suggested in the SIS manual. The mapped circuit is then used as an input to our optimization tool. The wire removal algorithm was run only once on each benchmark circuit, i.e., no repeated passes were performed and any single wire is examined only once; repeated passes did not increase substantially the number of wires removed to justify the time spent. The number of wires removed range from a low 5% to a maximum 30% of the original wire count. We ran the different heuristics on each of the benchmark circuits and the percentage of wires and nodes removed was recorded.

The optimized circuits (78 total) were then placed and routed using VPR [1], a place and route tool for FPGAs. We have used the timing-driven (based on Elmore delay) router in VPR. The FPGA routing architecture has single-length wire segments and two input/output pads per row and column. The FPGA architecture we have chosen is simple, as we wish to isolate the effects of wire removal/replacement from that of the FPGA architecture. The order of logic blocks, in circuit descriptions of both the original and optimized circuits, were forced to be the same (except of course, the redundant blocks removed), in order to reduce any side-effects of different input seen by VPR. The same experimental conditions are retained for the results in Section 3.

The results obtained in this section should not be construed as a manifestation of potential VPR drawbacks. Cur-

rently VPR produces the best results among various public domain tools in terms of quality of results. Our conjecture, drawn from our experiments, is that similar observations can be drawn if other place and route tools are used. Figure 3 shows the distribution of all the circuits after placement and routing. Symbol + indicates an increase, – a decrease and = no change in delay or width (measured by number of tracks per channel) of the routing channel. Of all the optimized circuits, after placement and routing, only 12.8% resulted in the reduction of critical path delay as well as the number of tracks (per channel) needed to route the circuit. On the other hand, 56.4% circuits had either negligible change ($< \pm 2\%$) in performance or were slow, disregarding the number of tracks (minimum) required to route. The results of Figure 3 clearly bring out the fact that the effects of placement-independent logic optimizations can be unpredictable.

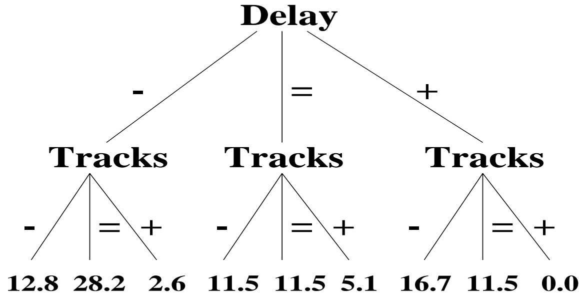

Figure 3. Distribution of placement and routing results.

The results concerning power dissipation have been encouraging. We have seen reduction in power across all the examples optimized (average 9.4%). Similar power reductions have been reported by [5]. The reduction in power can be attributed to two factors: (1) reduced circuit capacitance because of wire removal and (2) reduction in functional switching activity.

Table 1. Sample results to emphasize wire quality.

Circuit	*Wires removed*	*Nodes removed*	*Channel width*	*Delay* (ns)
C1355	35	5	6	43.6
	35	5	7	37.7
C3540	130	21	7	105.2
	127	22	7	88.9

As mentioned earlier, the percentage of wires removed was different depending on the heuristics applied. We have some interesting observations even in those cases where the number of wires removed was equal. Table 1 shows just a sample of two circuits optimized with two different heuristics. Similar observations have been made in other examples, but we do not list them all due to space limitations. For `C1355` the two heuristics removed the same number of wires (35) and nodes (5), but the resulting placed and routed circuits require different number of routing tracks and have different delays. The same is true for `C3540` (except for the number of tracks) when approximately the same number of wires and nodes are removed. This is due to the fact that different methods removed different wires. Hence, it is important to extract information about a wire before it is removed. As enough physical information is not available at the technology independent level, blind optimizations often lead to unpredictable results. Section 3 presents a solution to provide the necessary feedback to make *intelligent* wire removal and replacement.

3. Placement Dependent Logic Optimization

The results from the previous section suggest the need for physical information during logic optimization. Figure 4 shows the effect of using the placement of the original unoptimized circuit to route the optimized circuits. The majority of the circuits have the same performance characteristics as the original circuit. This is a clear indication of the importance of placement and the effect it has on final circuit characteristics. The rest of this section describes an algorithm that combines logic optimization (wire removal and replacement) and placement into a single step.

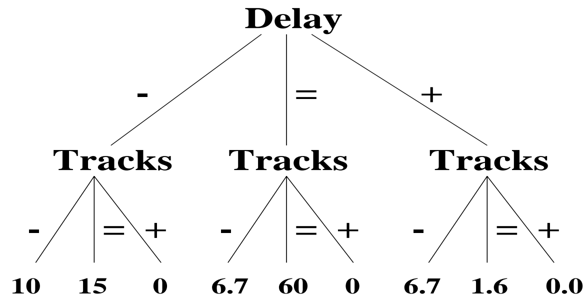

Figure 4. Distribution of placement and routing results.

Figure 5 shows the pseudo-code for the simulated annealing-based simultaneous logic and placement optimization algorithm. The generic simulated annealing algorithm for placement [1] is adapted to incorporate logic moves along with swapping of a pair of blocks. At every inner iteration, i.e., during iterations corresponding to a constant temperature, a logic move (wire removal or replacement) or a placement move is made based on a user-defined or automatic schedule. The user-defined schedule specifies the number of logic moves, typically 5-10, to be

made at each temperature. The automatic schedule has been determined empirically [1]. If a placement move is chosen, two blocks within a square of side $2 \cdot R_{limit}$ are chosen for swapping. R_{limit} ranges from 1 to the maximum number of rows or column in the FPGA. R_{limit} is decreased gradually as the temperature T is decreased. It initially covers the entire FPGA and at lower temperatures it is set such that only finer moves are performed. If a logic optimization move is chosen, the core (**Core**) of the basic wire removal algorithm of Figure 2 is invoked. The cost function, *linear congestion*, the weighted sum of the bounding boxes of the nets in the network, is updated after every move, either logic or placement. Individual nets are weighted according to their fanout count; the weights were determined empirically [1].

```
S = RandomPlacement();
T = InitialTemperature();
R_limit = InitialR();

while(ExitCriterion() == False) {
   while(InnerLoopCriterion() == False) {
      S_new = LogicOrPlaceMove(Network,S,
                                    R_limit);
      ΔC = Cost(S_new) - Cost(S);
      if(random(0,1) < e^(-ΔC/T)) {
         S = S_new;
      }
   }
   T = UpdateTemp();
   R_limit = UpdateR();
}
```

Figure 5. Simultaneous logic and placement optimization.

The logic move in the combined algorithm now has information about the current placement. The half perimeter length of the bounding box of a net, appropriately weighted by the fanout count, is used to sort nodes in the network. If switched capacitance heuristic is used instead, the circuit nodes are sorted according to the switching activity of a node times the half perimeter length of the bounding box. Fanout wires of the node with highest weight are then chosen as target wires. These fanout wires are sorted and processed according to the half perimeter length of their individual bounding box. If a target wire cannot be removed, it is examined for potential replacement. The details of wire replacement strategy we employ are explained with the help of Figure 6.

Consider the wire connecting blocks i and j. The half perimeter length of the bounding box connecting i and j is 3. The blocks that can potentially replace i (resulting in length less than 3, and hence, lower cost) are marked by a plus. We can search for a candidate block in this region or, on the other hand, we can intentionally increase the placement cost (non-greedy move) by searching for potential candidates in a larger region (unmarked blocks). Our algorithm incorporates logic moves that, like the placement moves, can move the cost in either direction and hence, potentially lead to a better solution.

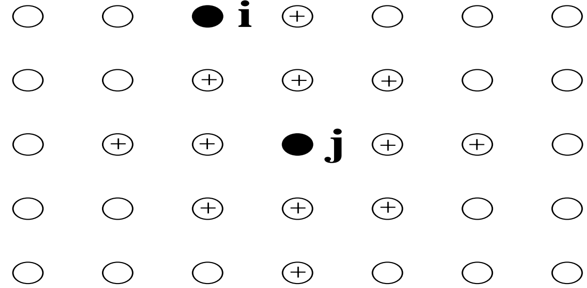

Figure 6. Search region for a replacement node.

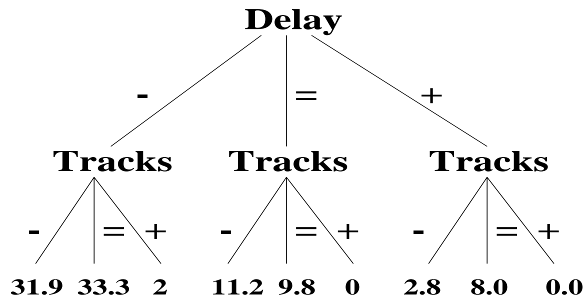

Figure 7. Distribution of placement and routing results.

The experimental results using the combined algorithm are presented in Figure 7. As mentioned earlier, the experimental conditions of Section 2.3 were retained. The combined algorithm leads to much better improvements when compared to Figure 3. 67.2% of the circuits now show performance gains as opposed to 43.6% due to placement independent optimization. Also, none of the circuits needed more tracks for the same performance (as opposed to 5.1%). Power reductions were higher (average 16.5%) than those of Section 2.3 due to better wire removal.

We performed an additional experiment to ascertain the advantages of simultaneous placement and logic optimization. The final optimized circuits (obtained after simultaneous placement and logic optimization) were again placed and routed, but this time ignoring the placement produced by the combined algorithm. The experiment revealed a distribution very similar to Figure 3, except for a further but marginal reduction in power dissipation and critical path delay due to extra wires removed.

Table 2 shows a sample of logic optimization results. The columns under *Original* show the original node and wire count. The number of wires, W, and nodes, N, removed

Table 2. Sample logic optimization results.

Circuit	*Original*		*PI*		*PD*	
	W	*N*	*W*	*N*	*W*	*N*
C1355	456	114	(32)71	(4)13	80	16
C1908	718	201	(110)91	(21)17	141	29
C3540	1597	462	(152)150	(26)24	179	26
C5315	2472	714	(277)313	(41)49	374	63
C880	530	142	(17)19	(3)3	38	4
alu2	504	141	(12)16	(1)2	38	2
apex6	888	242	(15)39	(2)3	68	5
apex7	298	85	(19)19	(4)4	29	4
des	5223	1493	(77)83	(6)6	250	6
f51m	136	39	(33)38	(8)9	44	10
rot	772	239	(10)17	(0)1	27	1
C2670	960	350	(146)156	(36)39	183	43
dalu	1705	471	(150)146	(16)21	189	30
frg2	1866	532	(45)69	(0)8	78	7
i9	1110	286	(147)145	(0)0	177	0
x3	1124	333	(27)65	(15)12	83	15
C7552	3490	1042	(524)600	(85)101	667	104
i10	3490	1004	(357)360	(22)22	471	35

after placement-independent optimization and placement-dependent optimization are shown under columns *PI* and *PD*, respectively. The table shows that the numbers of wires and nodes removed during placement-dependent optimization are consistently higher than those of placement-independent optimization. This is due to the fact that the order in which nodes are examined during placement-dependent optimization changes dynamically. On the contrary, during pure logic optimization, the nodes are chosen according to a particular cost function and hence, over repeated invocations of the algorithm of Figure 2, will very likely follow the same order. This observation is further corroborated by the pair of numbers reported under *PI* for each circuit. The first numbers, shown in brackets, are the best numbers obtained (among various heuristics applied) when nodes are sorted based on a specific cost function and the other when nodes are examined randomly. Our algorithm removes approximately 40% more wires and 27% more nodes than pure logic optimization technique.

4. Conclusions

In this paper, we have presented a technique to perform simultaneous logic and placement optimization for circuits mapped to FPGAs. Our algorithm combines logic and placement moves in a simulated annealing set up. The experimental results show that feedback from placement is very valuable in making informed choices during logic optimization. Logic optimization without such a feedback has little or no beneficial effect on final circuit characteristics. The combined algorithm produced circuits that have better performance and occupy less area when compared to placement-independent logic optimization. Also, the combined algorithm produced circuits that dissipated 7.8% less power than those produced by pure placement-independent logic optimization.

References

[1] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*, 1997.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[3] S.-C. Chang, K.-T. Cheng, N.-S. Woo, and M. Marek-Sadowska. Layout driven logic synthesis for FPGAs. In *Proceedings of the Design Automation Conference*, pages 308–313, San Diego, CA, June 1994.

[4] J. Cong and Y. Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in look-up table-based FPGA designs. *IEEE Transactions on Computer-Aided Design*, 13(1):1–12, Jan. 1994.

[5] J.-M. Hwang, F.-Y. Chiang, and T. T. Hwang. A re-engineering approach to low power FPGA design using SPFD. In *Proceedings of the Design Automation Conference*, pages 722–725, San Francisco, CA, June 1998.

[6] B. Kumthekar, L. Benini, E. Macii, and F. Somenzi. In-place power optimization for LUT-based FPGAs. In *Proceedings of the Design Automation Conference*, pages 718–721, San Francisco, CA, June 1998.

[7] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method—design of logic networks based on permissible functions. *IEEE Transactions on Computers*, C-38(10):1404–1424, Oct. 1989.

[8] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, Cambridge, MA, Oct. 1992.

[9] N. Shenoy, M. Iyer, R. Damiano, P. Thilking, K. Harer, and H.-K. Ma. A robust solution to the riming convergence problem in high-performance design. Presented at IWLS99, Lake Tahoe, CA, June 1999.

[10] S. Sinha and R. K. Brayton. Implementation and use of SPFDs in optimizing boolean networks. In *Proceedings of the International Conference on Computer-Aided Design*, pages 103–110, San Jose, CA, Nov. 1998.

[11] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[12] S. Yamashita, H. Sawada, and N. Nagoya. A new method to express functional permissibilities for LUT based FPGAs and its applications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 254–261, San Jose, CA, Nov. 1996.