Design, Synthesis and Evaluation of Heterogeneous FPGA with Mixed LUTs and Macro-Gates

Yu Hu
Electrical Engineering Dept.
Univ. of California Los Angeles
Los Angeles, CA 90095
hu@ee.ucla.edu

Satyaki Das
Xilinx Research Lab
2100 Logic Dr.
San Jose, CA 95124
satiyaki.das@xilinx.com

Lei He
Electrical Engineering Dept.
Univ. of California Los Angeles
Los Angeles, CA 90095
lhe@ee.ucla.edu

ABSTRACT

Small gates, such as AND2, XOR2 and MUX2, have been mixed with lookup tables (LUTs) inside the programmable logic block (PLB) to reduce area and power and increase performance in FPGAs. However, it is unclear whether incorporating macro-gates with wide inputs inside PLBs is beneficial. In this paper, we first propose a methodology to extract a small set of logic functions that are able to implement a large portion of functions for given FPGA applications. Assuming that the extracted logic functions are implemented by macro-gates in PLBs, we then develop a complete synthesis flow for such heterogeneous PLBs with mixed LUTs and macro-gates. The flow includes a cut-based delay-optimal technology mapping, a mixed binary integer and linear programming based area recovery algorithm to balance the resource utilization of macro-gates and LUTs for area-efficient packing, and a SAT-based packing. We finally evaluate the proposed heterogeneous FPGA using the newly developed flow and show that mixing LUT and macro-gates, both with 6 inputs, improves performance by 16.5% and reduces logic area by 30% compared to using merely 6-input LUTs.

1. INTRODUCTION

The popular island style FPGA architecture [1] consists of *programmable logic blocks* (PLBs) embedded in routing channels. The logic element within the PLB can be a lookup table (LUT) [3], programmable logic array (PLA) [4], or macro-gate (e.g. AND gates and multiplexers) [5]. These logic elements offer a spectrum of trade-offs between functionality and costs in terms of area, power and delay. For instance, a circuit can be implemented by fewer K-input LUTs than by K-input macro-gates, while a K-input macro-gate requires smaller silicon area and has lower propagation delay than a K-input LUT. The PLB is *heterogeneous* if it consists of different types of logic elements, otherwise it is *homogeneous*. In this paper, we assume that heterogeneity exists only inside a PLB while the structures of all PLBs are identical, and study the impact of heterogeneous PLBs.

Recent work has shown when uniform LUTs are used for the PLB, a larger LUT size increases performance [6, 7], but it reduces the LUT pin utilization rate. For example, mapping IWLS'05 benchmarks [8] using 6-input LUTs and the Berkeley ABC mapper [22], we find that over 60% of the LUTs use less than five inputs. Initial studies in the literature have suggested that mixed-sized LUTs [6, 9, 10, 25], mixing LUTs and PLAs [11] inside the PLB, or PLBs with hard-wired connections [26] may improve logic density. In addition, commercial FPGAs [6] have benefited from small macro-gates (e.g., XOR2 and MUX2) inside the PLB. However it is unclear whether incorporating macro-gates with wide inputs inside PLB is beneficial. The first contribution of this paper is to propose a methodology to extract a small set of logic functions that are able to implement a large portion of functions for given FPGA applications. Assuming that the extracted logic functions are implemented by macro-gates in PLB, we design a heterogeneous PLB consisting of both LUTs and macro-gates.

Effective and efficient synthesis tools are key enablers for the exploration of different architecture options. There are extensive studies (e.g., [12, 13, 14, 15, 16, 17]) on synthesis for homogeneous PLBs, but only limited research on synthesis for heterogeneous PLBs. [10] proposed heuristics to speedup the technology mapping for homogeneous PLBs and then extended them to consider heterogeneous PLBs with mixed LUT sizes. [18, 19] integrated Boolean Satisfiability solvers into re-synthesis to deal with macro-gates in heterogeneous PLBs. However, the high time complexity prohibits exploring complicated heterogeneous PLBs. The second contribution of this work is to develop an efficient logic synthesis flow for heterogeneous PLBs. The flow includes a cut-based delay-optimal technology mapping, a mixed binary integer and linear programming (MBILP) based area recovery algorithm to balance the resource utilization of macro-gates and LUTs for area-efficient packing, and a SAT-based packing. Using the newly developed flow, we evaluate the proposed heterogeneous PLB, and show that mixing LUT and macro-gates, both with 6 inputs, improves performance by 16.5% and reduces logic area by 30% for IWLS'05 benchmarks [8] when compared to using 6-input LUTs.

The rest of this paper is organized as follows. Section 2 presents the methodology to design heterogeneous PLBs with mixed LUTs and macro-gates. Section 3 describes the improved technology mapping and post-mapping area recovery algorithms to deal with the proposed FPGA architecture. Section 4 proposes a flexible SAT-based packing algorithm. The architecture evaluation results are given in Section 5, and the paper is concluded in Section 6. To the best of our knowledge, this paper is the first systematic study of the logic synthesis flow for FPGAs consisting of heterogeneous PLBs with wide-input macro-gates.

2. HETEROGENEOUS PLB DESIGN

The key step in designing a heterogeneous PLB is to extract a small set of logic functions that are able to implement a large portion of functions in given FPGA applications. In this section, we discuss how to extract such a set of logic functions by performing logic function ranking, and then present our macro-gate design.

2.1 NPN-Class Diagram (NCD)

To rank the logic functions extracted from the training FPGA application set, we need to identify two important relationships i.e., NPN-equivalence and inheritance equivalence, for every two logic functions.

DEFINITION 1. ***(NPN Equivalence)*** *Two boolean functions, F and G, belong to the same NPN-class (NPN-equivalent) if F can be derived from G by negating (N) and permuting (P) inputs and negating (N) the output. [20]*

For example, $F = ab + c$ and $G = a'c + b$ are NPN-equivalent. Although there are $2^{2^4} = 65536$ different 4-variable functions, only 222 of them are NPN independent (i.e., NPN-inequivalent to each other). In the rest of this paper, we use the term "NPN-class" to denote a set containing only NPN-independent elements. In this work, we limit the programmability of a macro-gate with input/output negation, which provides a good trade-off between flexibility and cost (performance and area). For an N-input macro-gate with input/output negation, $N+1$ memory cells (i.e., configuration bits) are need rather than 2^N memory cells for an LUT with N inputs (LUT-N).

DEFINITION 2. ***(Inheritance Equivalence)*** *For logic function $A = F_1(a_1, \cdots, a_n)$ and $B = F_2(b_1, \cdots, b_m)$, where $0 \leq m < n$. A is inheritance equivalent to B iff $\exists\ a_i$, s.t. $A(a_1, \cdots, a_i = 1, \cdots, a_n)$ or $A(a_1, \cdots, a_i = 0, \cdots, a_n)$ and B are NPN-equivalent.*

In the other words, A and B are NPN-equivalent if we fix one of the inputs of A as logic zero or one, i.e., B is a cofactor of A. For example, $f_1(a, b, c, d) = abcd$ is inheritance equivalent to $f_2(a, b, c) = abc$ by fixing input d as one. In fact, f_1 is a 4-input AND gate, which can be used to implement a 3-input AND gate (i.e. f_2). The inheritance equivalence of the full set of NPN-class with up to N inputs can be represented by the following NPN-class diagram (NCD).

DEFINITION 3. ***NPN-Class Diagram (NCD)*** *is a directed acyclic graph (DAG), where each node in level N is the representative[1] of a N-input NPN-Class, and each edge from node A to node B indicates that function A is inheritance equivalent to B, i.e. A can implement B's functionality.*

Note that Inheritance Equivalence is asymmetric, e.g., f_1 is inheritance equivalent to f_2 but the reverse is not true. Therefore, NCD is a DAG as no edge is from lower level node to high level node. Figure 1 shows the NCD for all 3-input functions.

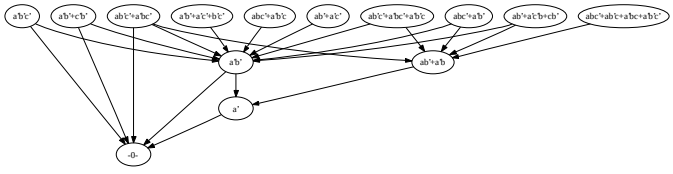

Figure 1: NCD for 3-input functions

[1]The representative (canonical form) of a N-input NPN-Class can be selected based on different rules, but the NCD is always applicable.

2.2 Utilization NPN-Class Diagram (UND)

To extract N-input macro-gate logic functions from the training FPGA application set, we first map those applications by exclusive LUT-N architecture and then analyze all logic functions that are mapped into LUTs. Note that NCD stores all different function categories and their relationship within a DAG, which makes it extremely efficient to explore some interesting properties of an application. To represent the functions implemented by each particular application based on NCD, we present the Utilization NPN-Class Diagram (UND), which is a sub-graph of NCD in addition to the weights associated with each node. UND can be defined recursively as follows.

DEFINITION 4. ***(Utilization NPN-Class Diagram (UND))*** *For a particular application D, unitilization NPN-Class diagram $\mathcal{U}_D$ of D is a sub-graph of NCD. If a function F is implemented by at least one of the LUTs in D, the NCD node that corresponds to the NPN-Class of F should be added into $\mathcal{U}_D$. If a NCD node is present in $\mathcal{U}_D$, all of its fanout nodes and edges should be added into $\mathcal{U}_D$ recursively. Each node is associated with a 3-tuple $\Phi(f, n, c)$, where f is the functionality description of this NPN-Class, n and c are the implemented frequency and implementation capability (will be defined later), respectively.*

DEFINITION 5. ***(Implemented Frequency (IF))*** *For a NPN-Class function f presented in a mapped application, the implemented frequency IF_f of logic function f is the number of LUTs which implement logic function f.*

DEFINITION 6. ***(Implementation Capability (IC))*** *For a NPN-Class function f presented in the UND of a mapped application, the implementation capability IC_f of f is calculated by the following equation:*

$$IC_f = \frac{\sum_{\forall v \in fanout\ cone\ of\ f} IF_v}{\sum_{\forall u \in UND} IF_u} \quad (1)$$

Intuitively, IC_f of NPN-Class function f indicates the portion (in terms of percentage) of logic functions in the application that can be implemented by f.

Figure 2 shows the UND for a small application that is mapped by LUT-3. There are totally 12040 functions implemented by LUTs in this application, and only three 3-input NPN-classes are presented. An interesting observation from Figure 2 is that the IC for function $g = ab'c + a'bc' + a'b'c$ is 55% while its IF is only 1120 (less than 10% of 12040 total functions). In fact, it has a child node $ab' + a'b$ whose IF is 4410 (36% of 12040 total functions), if we look into the fanout cone (shadowed area) of node $ab'c + a'bc' + a'b'c$.

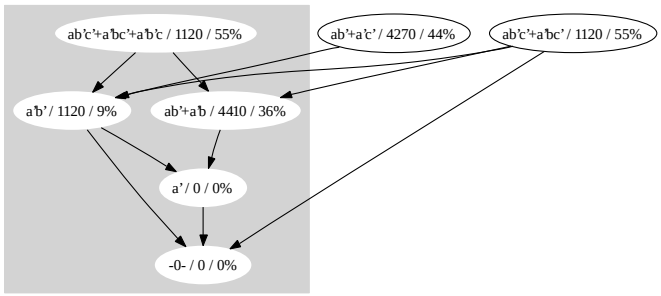

Figure 2: An example UND

As we are interested in the minimal number of logic gates to cover all logic functions in an application, i.e., minimal functional covering (MFC), we propose the following theorem to calculate the MFC very efficiently.

THEOREM 1. *The number of minimal functional covering (MFC) in application D is equal to the number of all primary input nodes (with no parent nodes) of UND for D.*

The proof of this theorem is straightforward.

2.3 Logic Function Ranking Metrics

Our objectives for ranking logic functions are (i) using fewer logic functions to cover larger subset of all functions presented in the mapped application, and (ii) keeping high utilization of input pins. In fact, there is a trade-off between those two objectives. Figure 3 shows UNDs for four toy applications. Each node is labeled by its functionality and IF. In case (a), function $ab' + a'c'$ appears once and ab appears 10 times. $ab' + a'c'$ should have a higher rank than ab if the objective is to minimize the number of NPN-classes that can cover all functions. On the other hand, if we are more interested in pin utilization, ab is a better choice, as 10 gates with function ab' only use 2/3 pins if $ab' + a'c'$ is used to implemented all functions in case (a).

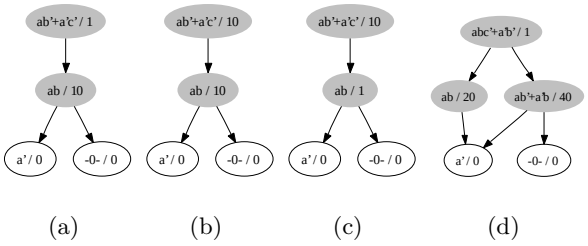

Figure 3: Illustration for the trade off among various metrics

In this work, we propose a new metric, Utilization Weighted Implementation Capability (UWIC), which is a deterministic analytical metric of NPN-Class functions and leads to a good trade-off between logic function covering and pin utilization.

DEFINITION 7. ***Utilization Weighted Implementation Capability (UWIP)*** *of NPN-Class function f is*

$$UWIP_f = \sum_{\forall v \in fanout\ cone\ of\ f} \frac{num\_input(v)}{num\_input(f)} \cdot IF_v \quad (2)$$

We calculate the UWIP values for each case in Figure 3 as follows.

1. $UWIP_{ab} = 10 \times 1.0 = 10, UWIP_{ab'+a'c'} = 10 \times 2/3 + 1 \times 1.0 = 7.7$, so we choose ab.
2. $UWIP_{ab} = 10 \times 1.0 = 10, UWIP_{ab'+a'c'} = 10 \times 2/3 + 10 \times 1.0 = 16.7$, so we choose $ab' + a'c'$.
3. $UWIP_{ab} = 1 \times 1.0 = 1, UWIP_{ab'+a'c'} = 1 \times 2/3 + 10 \times 1.0 = 10.3$, so we choose $ab' + a'c'$.
4. $UWIP_{ab} = 20 \times 1.0 = 20, UWIP_{ab'+a'b} = 40 \times 1.0 = 40, UWIP_{abc'+a'b'} = 1 \times (20+40) \times 2/3 + 1 \times 1.0 = 41$, so we choose $abc' + a'b'$.

It shows that the UWIP values are consistent with the good trade-off between logic function covering and pin utilization that we just observed.

2.4 UND for Wide Input Functions

To explore less-than-5-input functions, we can build 4-input NCD once and use a lookup table to store the NPN-class of every logic function. For each of the training applications that are mapped by LUT-4, we can find its NPN-class by the lookup table and create or label the node in UND. However, this procedure becomes prohibitively expensive since we cannot afford to pre-construct NCD for more than 4-input functions by exhaustively examining NPN-equivalence for even all 5-input functions ($2^{2^5} = 4294967296$).

Practically, one can build a *partial UND* by on-line checking NPN-equivalence for over-5-input functions, which can be performed efficiently by the method proposed in [20]. When a new node representing a N-input ($N \geq 5$) NPN-class function is inserted in the partial UND, only those (N-1)-input functions that are inheritance equivalent to it are inserted/labeled in the partial UND, instead of performing insertion recursively. Partial UND is a good approximation of UND and all methods for UND manipulation are applicable for partial UND. In experiments, we find that the total number of all 6-input NPN-classes presented in all IWLS'05 benchmarks [8] is less than 5000 and only 167 of them are present more than 1% out of all functions. Therefore, the size of partial UND can be well controlled in practice.

2.5 Macro-Gate Design

The logic function extraction and ranking framework proposed above is implemented by the mix of Perl and C on Berkeley ABC [22] platform. Using IWLS'05 benchmarks [8] as the training set, the following 6-input NPN-classes with the highest ranks are found as the candidates of the macro-gates to be added into the FPGA PLB.

$$g_1(a,b,c,d,e,f) = abcdef \quad (3)$$
$$g_2(a,b,c,d,e,f) = a'b'c' + bcf' + bc'd' + b'ce' \quad (4)$$
$$g_3(a,b,c,d,e,f) = ab'cd'e + bcef + def \quad (5)$$
$$g_4(a,b,c,d,e,f) = ab' + a'cd' + b'c' + e' + f' \quad (6)$$

Combining these four gates with input/output negation, we can implement 23% 6-input functions, 34% 5-input functions, 60% 4-input functions, 69% 3-input functions and 98% 2-input functions on average for IWLS'05 benchmarks. Overall 50% functions can be implemented by this macro-gate. The structure of the macro-gate is shown in Figure 4. In our experiments, we incorporate this macro-gate into the FPGA PLBs and assume that there exist one LUT and one such macro-gate in each PLB.

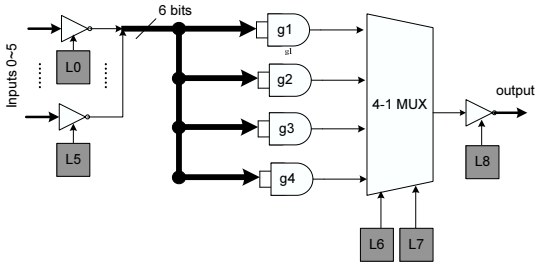

Figure 4: Architecture of the macro-gate

3. TECHNOLOGY MAPPING

Given the functions of the macro-gates to be built into FPGA PLBs, a technology mapper is needed to make full use of these

macro-gates in terms of performance improvement and area reduction. A simple way to handle the heterogeneous architecture with both M-input LUTs and K-input macro-gates is the following 3-step algorithm. First, utilize the existing technology mapping algorithm which handles LUT based FPGAs to map the application by K-input LUTs, then replace those LUTs that can be implemented by K-input macro-gates, and finally remap the remaining sub-circuit by M-input LUTs. Although both mapping procedures in this 3-step algorithm are delay optimal, the final mapping result could be far from optimal since both mappers have no global views.

3.1 Functional Cut Pruning

[18] presented a general algorithm for heterogeneous FPGA technology mapping, but it is not scalable to large macro-gates. [17] extended the traditional cut based technology mapping algorithm [10] (for LUT only FPGAs) to achieve delay optimality for macro-gate based FPGAs[2] and proposed a novel concept, i.e., factor cuts, to improve the scalability of the algorithm. In [17], the truth table of the NPN-classes that can be implemented by the macro-gates (with M inputs) are pre-calculated and stored in a lookup table. Then the mapping is done in two passes. The first pass is a forward topology traversal for enumerating all M-feasible cuts and keeping only those that can be found in the lookup table storing the truth tables of macro-gates. The second pass is a backward topology traversal to select the best cut for every node under the delay target and area constraints.

For technology mapping to heterogeneous PLBs with both LUTs and macro-gates, we adopt the same framework presented in [17] by labeling cuts as macro-gates or LUTs based on their logic functions. However, if $M \gg K$, i.e., the inputs of macro-gates are much more than the inputs of LUTs in the target architecture, cut enumeration is still expensive even with factor cuts [17]. In our experiments, we observe that certain P-cuts ($M > P \geq K$) can be pruned by matching their functions with the cofactors of macro-gate functions in the forward traversal process and preventing their further propagation. We pre-calculate and store all $M-1$ to K cofactors for each function of the macro-gate. Empirically, if a P-cut cannot match any of these cofactors, its extension[3] and itself will unlikely be implemented by the macro-gate. Figure 5 shows a fragment of the And-Invert graph (AIG) [17], where the dot-arrow edge indicates an inverter. The shadow shows a 4-feasible fanout free cut [10] $C_y = (b, c, d)$ for node y. The function of cut C_y is $f_y = b(d' + c')$. We can extend cut C_y with node z to obtain a new 4-feasible cut $C_x = (b, c, d, z)$ for node x whose function is $f_x = b(d' + c')z$. Suppose function f_y is not a cofactor of any macro-gate functions, which indicates that $f_x = f(y) \wedge z$ cannot be implemented by macro-gates if f_z is not a constant 0. In this case, we can prune cut C_y. On the other hand, if z provides a controlling value for x, i.e., $f_z \equiv 0$, cut C_y can still be pruned since the function of C_y becomes an Observability Don't-Cares (ODC) term. Based on the above observation, the following empirical rule is proposed for cut pruning.

RULE 1. *(Functional Cut Pruning) For a given node v, if the function of a fanout free cut [10] C_v rooted at node v is not a cofactor of any macro-gate functions, cut C_v is pruned.*

Our experiments show that the proposed functional cut pruning rule reduces more than 50% cuts when M=6 and K=4, and preserves both delay and area quality compared to technology mapping without functional pruning. Note that this rule is extremely helpful when $M \gg K$ since a large portion of more-than-K-input cuts will be pruned. Also the functional pruning rule is orthogonal to the factor cut pruning [17] and it can be applied along with factor cut pruning to further reduce runtime.

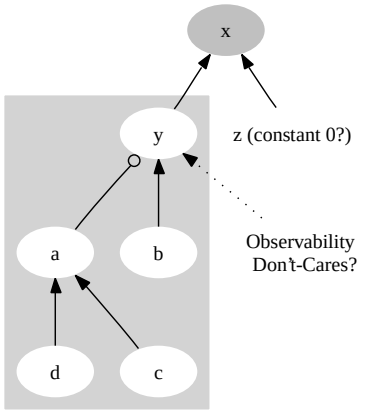

Figure 5: An AIG fragment with a fanout free cut (rooted at y)

3.2 Architecture-Aware Area Recovery

For the target PLB with C_L LUTs and C_M macro-gates, suppose there are N_L LUTs and N_M macro-gates in the mapped circuit, the logic area (i.e., the number of PLBs) after the ideal packing should be

$$\begin{aligned}\Phi(N,\alpha) &= \max(\frac{N_M}{C_M}, \frac{N_L}{C_L}) \\ &= \frac{N}{C_L} \cdot \max(\frac{\alpha}{\beta}, 1-\alpha)\end{aligned}$$

where $N = N_M + N_L$, $\beta = C_M/C_L$ and $\alpha = N_M/N$.

Since C_L and β are constant for a given architecture, it is easy to show

$$\min_{\forall \alpha} \Phi(N,\alpha) = \frac{N}{C_L} \cdot \frac{1}{1+\beta} \quad (7)$$

$$\alpha^* = \frac{\beta}{1+\beta} \quad (8)$$

For a given architecture, (7) provides a lower bound of the number of PLBs for a mapped result, where parameters N and α decide the tightness of the packing. (8) show that the tightest packing can be achieved when the LUT-MG ratios[4] of the given architecture ($\frac{\beta}{1+\beta}$) and the mapped result (α) are equal.

For example, suppose the target architecture has one LUT and one macro-gate within a PLB, i.e., the LUT-MG ratio is 1:1. If we can achieve the same ratio in the mapped result, a tight packing is expected. In fact, we can adjust the area weight assigned to LUTs and macro-gates in the technology mapper, which shows significant impact in the LUT-MG ratio of the mapped result. Figure 6 shows the average number of LUTs and macro-gates in the mapped results for 20 IWLS 2005 benchmarks with different area weight assignments for a 6-input LUT and a 6-input macro-gate, i.e., 1:1, 1:0.95, 1:0.9, 1:0.8, 1:0.5 and 1:0.1. The LUT-MG ratio can be clearly seen in this figure. To achieve a tight packing for the target

[2] The macro-gate in [17] was built by three LUTs (without the logic gates) and the homogeneous PLBs were assumed.

[3] We use the way proposed in [10] to generate all the cuts by extending cuts in topology order.

[4] LUT-MG ratio is the number of LUTs divided by the number of macro-gates in an application.

architecture with LUT-MG ratio 1:1, we first perform a binary search to find the best area weight assignment which gives a small N and an α close to the target LUT-MG ratio. From Figure 6, we see that the total number of LUTs and macro-gates (N in (7)) increases dramatically if an extremely small weight is assigned to macro-gates (see the bar for 1:0.5 and 1:0.1) since the mapper is biased. We find that 1:0.95, 1:0.9 and 1:0.8 all give good trade-offs between N and α under the target LUT-MG ratio but it is hard to further improve resulting LUT-MG ratio, β, by simply adjusting area weight.

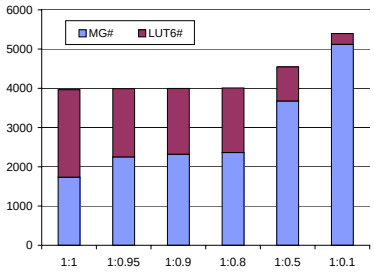

Figure 6: Impact of area weight on LUT-MG ratio

3.2.1 MBILP Formulation

Given the mapped result after binary search, the total number of LUTs and macro-gates, N, and delay target, T, are fixed. Motivated by the timing slack budgeting [21], we can reassign macro-gates and LUTs in such a way that the resulting LUT-MG ratio is sufficiently close to the target LUT-MG ratio while preserving the delay target. The combinational portion of the mapped result is represented in a DAG $G = (V, E)$, where $\forall v_j \in V$ is mapped to an LUT or a macro-gate. Without loss of generality, we assume that the intrinsic delay of an LUT is ΔD larger than a macro-gate. In this case, the binary search should be able to find an LUT-MG ratio, α, which is slightly smaller than the target LUT-MG ratio, $\beta/(1+\beta)$. To balance the number of LUTs and macro-gates according to β, certain macro-gates should be re-mapped as LUTs. For each node v_j that is mapped as a macro-gate, if its input number is not larger than the LUT size, it can be re-mapped as an LUT without changing the functionality. All such nodes are stored in set V_m. A binary variable m_j indicates if node v_j can be re-mapped as an LUT for the given timing slack b_j. The objective is to minimize the gap between the mapped macro-gate number $\sum_{\forall v_j \in V_m(G)} m_j$ and the ideal number $\frac{1}{1+\beta} \cdot N$. The overall problem can be formulated as a mixed binary integer and linear programming (MBILP) as follows.

$$\min \quad |\sum_{\forall v_j \in V_m(G)} m_j - \tfrac{1}{1+\beta} \cdot N| \tag{9}$$

$$\text{s.t.} \quad m_j \leq \tfrac{b_j}{\Delta D}, \forall v_i \in V_m(G) \tag{10}$$

$$m_j \in \{0,1\}, \forall v_i \in V_m(G) \tag{11}$$

$$a_i + d_j + b_j \leq a_j, \forall e(i,j) \in E(G) \tag{12}$$

$$b_i \geq 0, \forall v_i \in V(G) \tag{13}$$

$$a_i \leq T, \forall v_i \in V(G) \tag{14}$$

where a_i and d_j are the arrival time and intrinsic delay for node v_i, respectively. All the other variables have been explained above. Note that the absolute operator in the objective function can be easily transformed into linear form by introducing an auxiliary variable t as follows.

$$\min \quad t \tag{15}$$

$$\text{s.t.} \quad \sum_{\forall v_j \in V_m(G)} m_j - \tfrac{1}{1+\beta} \cdot N \leq t$$
$$\sum_{\forall v_j \in V_m(G)} m_j - \tfrac{1}{1+\beta} \cdot N \geq -t$$
$$t \geq 0 \tag{16}$$

3.2.2 Example

Figure 7 shows a circuit which is pre-mapped by LUTs and macro-gates with the same inputs. The delay of an LUT and a macro-gate is 5 units and 4 units, respectively. The rectangular nodes represent LUTs and the elliptical nodes represent macro-gates. The numbers in each node denote the arrival time and required time of the node given that the required time in PO is equal to the clock period.

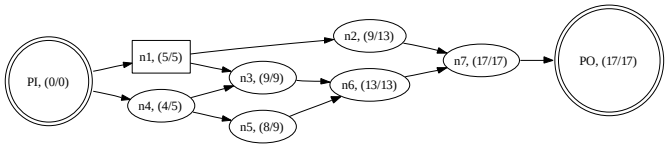

Figure 7: The pre-mapped circuit

Based on the MBILP formulation presented in Subsection *3.2.1*, the MBILP form for this area recovery problem can be expressed as follows. Shown in (12), arrival time constraints are listed according to all edges.

$$\begin{aligned} &a_{PI} + 5 + b_1 \leq a_1, \quad a_{PI} + 4 + b_4 \leq a_4, \quad a_1 + 4 + b_2 \leq a_2, \\ &a_1 + 4 + b_3 \leq a_3, \quad a_4 + 4 + b_3 \leq a_3, \quad a_4 + 4 + b_5 \leq a_5, \\ &a_2 + 4 + b_7 \leq a_7, \quad a_3 + 4 + b_6 \leq a_6, \quad a_5 + 4 + b_6 \leq a_6, \\ &a_6 + 4 + b_7 \leq a_7, \quad a_7 \leq a_{PO} \end{aligned} \tag{17}$$

Based on (13), the following constraints guarantee that the timing slack budget is nonnegative.

$$b_i \geq 0, \quad i = \{1, \cdots, 7\} \tag{18}$$

Based on (14), the critical path delay is preserved by the following constraints.

$$a_{PO} \leq T, \quad a_{PI} = 0 \tag{19}$$

Based on (10), the following constraints leverage timing slack with resource remapping.

$$(5-4) \cdot m_i \leq b_i, \quad m_i = \{0,1\}, \quad i = \{2, \cdots, 7\} \tag{20}$$

Since there are equal numbers of LUTs and macro-gates in the target PLB, the ideal number of macro-gates is $N/2 = 7/2$. Therefore, the objective is to minimize the gap between the number of macro-gates $(m_2 + \cdots + m_7)$ and 7/2. We write it as a linear form based on (16) as follows.

$$\begin{aligned} min \quad & t \\ s.t. \quad & m_2 + \cdots + m_7 - 7/2 \leq t, \\ & m_2 + \cdots + m_7 - 7/2 \geq t \end{aligned} \tag{21}$$

Solving the MBILP problem with objective (21) and constraints (17), (18), (19) and (20), the solutions are as follows.

$$\begin{aligned} a_1 = 5, a_2 = 10, a_3 = 9, a_4 = 5, a_5 = 9, \\ a_6 = 13, a_7 = 17, a_{PO} = 17, a_{PI} = 0, \\ b_3 = b_5 = b_6 = b_7 = 0, b_2 = b_4 = 1, \\ m_3 = m_5 = m_6 = m_7 = 0, m_2 = m_4 = 1 \end{aligned}$$

This solution corresponds to a set of resource reassignment by remapping node 2 and node 4 to LUT-6 and the resulting application has 3 LUT-6 and 4 macro-gates, which achieves the best balance and can be packed into 4 PLBs. Compared to the original application, logic area is reduced by 30% (from 6 PLBs to 4 PLBs). The re-mapped circuit is shown in Figure 8.

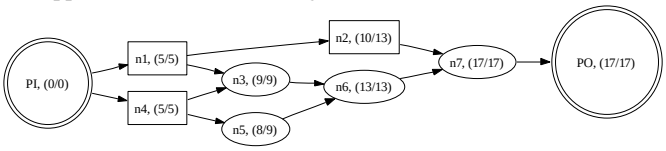

Figure 8: The re-mapped circuit after area recovery

3.2.3 Generalization

The above algorithm can be generalized to handle the case that the target architecture has more than one type of macro-gate. Without loss of generality, suppose there are two types of macro-gates in the target architecture, e.g., G_1 and G_2. The number of LUTs, G_1s and G_2s within a PLB is C_L, C_{g1} and C_{g2}, respectively. The intrinsic delay of LUTs, G_1s and G_2s within a PLB is D_{lut}, D_{g1} and D_{g2}, respectively. The objective is to remap the logic elements so that the number of LUTs, G_1s and G_2s in the resulting application is as close to $C_L : C_{g1} : C_{g2}$ as possible.

For each node (logic element) n_i, if n_i can be implemented by an LUT (i.e., fan-in number of this node is smaller or equal to LUT inputs), we associate a binary variable u_i with it, where u_i = 1 means that this node is remapped to an LUT, otherwise it is remapped by G_1 or G_2. In the same manner, we associate binary variables g_{1i} and g_{i2} to indicate if node n_i is mapped to G_1 or G_2. The following constraints guarantee that only one mapping is accepted.

$$u_i + g_{1i} + g_{i2} = 1 \tag{22}$$

To leverage the timing slack with the resource assignment solution, we have the following constraints according to the current mapping of node i. For concise presentation, we use $n_i \models G$ to denote that node i can be implemented by device G (could be an LUT or macro-gate).

Case 1. Node i has been mapped by LUT at present.

$$(D_{g1} - D_{lut}) \cdot g_{1i} \leq b_i \quad \text{, if } n_i \models G_1 \tag{23}$$
$$(D_{g2} - D_{lut}) \cdot g_{i2} \leq b_i \quad \text{, if } n_i \models G_2 \tag{24}$$

Case 2. If node i has been mapped by G_1 at present.

$$(D_{lut} - D_{g1}) \cdot u_i \leq b_i, \text{, if } n_i \models \text{LUT} \tag{25}$$
$$(D_{g2} - D_{g1}) \cdot g_{i2} \leq b_i, \text{, if } n_i \models G_2 \tag{26}$$

Case 3. If node i is mapped by G_2 at present, we have the following constraints.

$$(D_{g1} - D_{g2}) \cdot g_{1i} \leq b_i, \text{, if } n_i \models G_1 \tag{27}$$
$$(D_{lut} - D_{g2}) \cdot u_i \leq b_i, \text{, if } n_i \models \text{LUT} \tag{28}$$

The non-negativity constraints for timing slack budget is not needed since we may have the case that $D_{g1} < D_{g2}$ and node i is remapped from G_2 to G_1. All other timing constraints are the same as the above example.

The objective function is represented as follows.

$$\begin{aligned} min \quad & W_u \cdot |\sum u_i - \tfrac{N \cdot C_L}{C_L + C_{g1} + C_{g2}}| + \\ & W_1 \cdot |\sum g_{1i} - \tfrac{N \cdot C_{g1}}{C_L + C_{g1} + C_{g2}}| + \\ & W_2 \cdot |\sum g_{i2} - \tfrac{N \cdot C_{g2}}{C_L + C_{g1} + C_{g2}}| \end{aligned} \tag{29}$$

which can be linearized as follows.

$$min \quad t_u + t_1 + t_2 \tag{30}$$
$$s.t. \quad \tfrac{t_u}{W_u} \leq \sum u_i - \tfrac{N \cdot C_L}{C_L + C_{g1} + C_{g2}} \leq \tfrac{t_u}{W_u} \tag{31}$$
$$\tfrac{t_1}{W_1} \leq \sum g_{1i} - \tfrac{N \cdot C_{g1}}{C_L + C_{g1} + C_{g2}} \leq \tfrac{t_1}{W_1} \tag{32}$$
$$\tfrac{t_2}{W_2} \leq \sum g_{i2} - \tfrac{N \cdot C_{g2}}{C_L + C_{g1} + C_{g2}} \leq \tfrac{t_2}{W_2} \tag{33}$$
$$t_u \geq 0, t_1 \geq 0, t_2 \geq 0 \tag{34}$$

where W_1, W_2 and W_u are the weight to indicate which resource is the most crucial one that needs to be balanced.

4. SAT BASED FLEXIBLE PLB PACKER

Given the mapped application, the following step is to pack the logic elements, such as LUTs, FFs and macro-gates, into FPGA PLBs. This problem essentially solves a clustering problem since we want to group circuit elements that are connected together in the same PLB so that wire lengths are reduced and routing congestion is avoided. Typically, a packer (e.g. *T-VPack* [1]) uses heuristics to determine what circuit elements should be clustered together. Then each of the clusters is checked to see if it can really fit into a FPGA PLB according to certain constraints. In T-VPack, the clustering constraints include the number of PLBs, in/out pins and clocks of each cluster. However the PLB architecture of recent commercial FPGAs is more complicated. In general, the clustering constraints that should be met for each cluster are as follows: (i) each circuit element in the cluster can be placed in certain location in the PLB, (ii) all interconnections within the cluster can be implemented in the PLB itself, and (iii) each outgoing/incoming signal is assigned to an appropriate PLB output/input pin.

However, the existing packing tools such as T-VPack hardcodes the architecture specification of a PLB and this check has to be written from scratch if the PLB architecture changes. This is time consuming and needs to be debugged carefully to avoid bugs.

Different from [18] and [19], we solve the problem of validating PLB packing as a local place and route problem at the PLB level. It is a placement problem since multiple compatible sites may be present for each logic element in the cluster[5]. Also, for the chosen placement, all interconnections within the cluster must be routed in the PLB and all external connections must be connected to appropriate PLB input/output pins.

We use a satisfiability (SAT) solver to carry out this PLB packing validation. The tool takes a netlist description of the target PLB, the user netlist, and the netlist elements clustered into the PLB to generate a set of Boolean equations such that each solution to these equations represents a valid packing. Then a SAT-solver can be used to find a solution to this set of equations. If a solution exists then the PLB packing is valid and an assignment of the netlist elements to locations in the PLB and PLB input or output pins to nets is produced. If no solution exists, then the packing is invalid.

Now we briefly sketch the problem formulation with an example. Assume that we are attempting to fit the fragment described in Figure 9 (b) into the simplified Xilinx Spartan3-like PLB in Figure 9 (a). We have variables $X@A$, $X@B$ encoding placement of LUT X at either the site A or B. Now we generate the *exclusivity constraint* and *presence constraint* as follows.

$$(\neg X@A) \vee (\neg X@B) \tag{35}$$
$$X@A \vee X@B \tag{36}$$

Constraint (35) implies that if X is placed at A then it cannot be placed at B and vice versa. Constraint (36) means that X has to be

[5]For instance in Xilinx Spartan3 [5] PLB there are two LUTs.

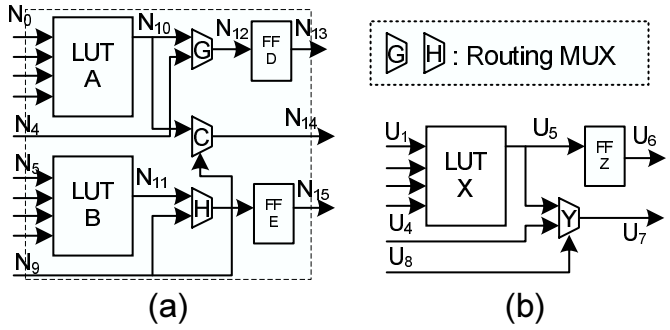

Figure 9: (a) Architecture of a FPGA PLB, (b) Subcircuit to be packed

placed at at least one of the sites A and B. The two equations above guarantee that X would be placed at exactly one location. Similar constraints would be generated for the flip flop Z and the possible locations D and E.

Nets are treated like logic element sites, which are encoded as Boolean variables, $U_i@N_j$ ($\forall i \in \{1 \cdots 8\}, j \in \{0 \cdots 15\}$). The mutual exclusion conditions apply on nets, as well, so that distinct nets in the user netlist cannot be present at any one net in the PLB.

We also have *input and output constraints* for all logic elements. One such output constraint would be

$$X@A \rightarrow U_5@N_{10}, \tag{37}$$

which guarantees that if X is placed at location A in the PLB, then the N_{10} net in the PLB carries the U_5 signal. Input constraints shall also be generated in an analogous fashion.

For routing multiplexers (like G and H) which are controlled by configuration logic we have variables that determine which input drives the output. So for G, we would have the variables, $G_{0\rightarrow\text{out}}$ and $G_{1\rightarrow\text{out}}$, and we need to add *exclusive driver constraints* (38) so that no output is driven by two inputs. We also need to add *routing constraints* (39) so that if a net is present at an input to a routing multiplexer and the appropriate control bit is active then the output has the correct net.

$$G_{0\rightarrow\text{out}} \rightarrow (\neg G_{1\rightarrow\text{out}}) \tag{38}$$

$$(G_{0\rightarrow\text{out}} \wedge U_5@N_{10}) \rightarrow U_5@N_{12} \tag{39}$$

Finally, extra variables are needed to encode the choice that some logic elements can act as route-throughs. An example of such an element is a lookup table. It can be programmed to pass one of its inputs out to the output and thus act like a routing multiplexer if it is not being used. This increases the routing flexibility if we need it. Corresponding to these extra variables, *route-through constraints* are generated. For example, some of the constraints generated for taking into consideration that LUT B might be used as a route-through are

$$(RT_B \wedge B_{0\rightarrow\text{out}}) \rightarrow (\neg B_{1\rightarrow\text{out}}) \tag{40}$$

$$(RT_B \wedge B_{0\rightarrow\text{out}} \wedge U_8@N_5) \rightarrow U_8@N_{11} \tag{41}$$

where RT_B encodes the choice as to whether LUT B is used as a route-through. Constraint (40) maintains that inputs 0 and 1 cannot drive the output simultaneously. There will be other constraints corresponding to mutual exclusion of every pair of inputs. Constraint (41) specifies that if input 0 does drive the output and if net N_5 is at input 0, then it would also appear at the output. Note that constraint (40) is just a representative constraint and other constraints related to the other input pins and to other possible nets will also be generated as needed.

Certain input pins of logic elements are swappable, i.e., they may be interchanged if it helps with producing a valid PLB. Variables are introduced to encode the choice of equivalent pins that have been swapped and *equivalent pin constraints* corresponding to these added.

Finally, *boundary constraints* forcing input (or output) nets to be assigned to at least one of the input (or output) pins are added. An example of such a constraint is

$$U_6@N_{13} \vee U_6@N_{14} \vee U_6@N_{15} \tag{42}$$

Once all the constraints are automatically generated from a topological description of the PLB graph and the logic elements in the user's netlist that we wish to pack into a PLB, we use a SAT-solver to check if they can all be satisfied simultaneously. If they are all satisfied, we can generate a valid placement and route from the solution that the SAT solver gives us. Translating the solution to an assignment of netlist elements to PLB locations, PLB input pins to nets and control bits in the routing multiplexers is easy. We construct the PLB based on all the variables that are TRUE in the satisfying assignment. In our example, the satisfying assignment could have the following variables be TRUE $U_1@N_0, U_2@N_1, \cdots, X@A, RT_B$, where $U_1@N_0$ means that the net U_1 is at the PLB input N_0, $X@A$ means that LUT X is placed at location A and RT_B means that LUT B is used as a route-through.

The principal advantage of this method is that we can validate PLB packing for different PLB architectures without any architecture specific code. This method can be especially useful for new architectures before specialized code has been written to validate packing and also to check correctness of the specialized code during regression testing. The method is also applicable for architecture exploration as it is infeasible to write specialized valid packing checkers for all candidate architectures.

5. ARCHITECTURE EVALUATION

5.1 Area and Delay Modeling

Suggested in [4], an LUT-4 can be implemented by 16 1-bit SRAM, 4 inverters and 15 2:1MUXes which are $16 \times 6 + 4 \times 2 + 15 \times 4 = 164$ transistors[6]. Following the FPGA area modeling in [4], an LUT-6 can be implemented by 64 1-bit SRM, 6 inverters and 63 2:1MUXes and a macro-gate in Figure 4 can be implemented by 9 1-bit SRM, 7 inverters, 3 2:1MUXes and 63 transistors (for implementing the embedded logic gates $g_1, \cdots, g_4$). Hence the number of transistors needed to implement a macro-gate and an LUT-6 is 143 and 648, respectively. Due to the space limit we omit the schematics of the implementation. The logic delay is modeled by the square root of the logic area. The area and delay models used in our experiments are summarized in Table 2.

	LUT-4	macro-gate	LUT-6
transistor number	164	143	648
normalized area	1.15	1.00	4.53
normalized delay	1.07	1.00	2.13

Table 2: Summary of the area and delay models

5.2 Experimental Results

We implement the improved mapping algorithms in Berkeley ABC platform [22] with C language and the SAT-based packer with

[6][4] assumes that a 1-bit SRAM, an inverter and a 2:1MUX needs 6, 2 and 4 transistors, respectively.

circuit	Logic depth				Estimated delay (normalized)				PLB#				Estimated area (normalized)			
	LUT4	LUT4 MG	LUT6	LUT6 MG	LUT4	LUT4 MG	LUT6	LUT6 MG	LUT4	LUT4 MG	LUT6	LUT6 MG	LUT4	LUT4 MG	LUT6	LUT6 MG
ac97_ctrl	4	4	3	4	4	4	6	4	4010	1739	2898	1523	4599	3733	13132	8424
aes_core	8	7	6	9	8	7	12	10	8959	4147	3912	2932	10275	8903	17727	16218
des_area	10	10	7	10	10	10	14	11	2153	1004	968	674	2469	2154	4386	3728
des_perf	6	6	4	7	6	6	8	7	32371	15652	6872	4863	37125	33601	31140	26900
ethernet	10	9	7	9	10	9	14	12	5047	2438	3953	2215	5788	5234	17913	12252
i2c	5	4	4	4	5	4	8	6	392	188	265	144	450	404	1201	794
mem_ctrl	11	9	8	9	11	9	16	12	3687	1731	2892	1413	4228	3715	13105	7813
pci_bridge32	9	8	7	8	9	8	14	11	6596	3156	5413	2911	7565	6774	24529	16102
pci_spoci_ctrl	6	5	5	5	6	5	10	7	372	173	251	137	427	370	1137	755
sasc	3	3	2	4	3	3	4	4	206	87	147	85	236	186	666	470
simple_spi	4	4	3	4	4	4	6	4	295	121	200	113	338	259	906	625
spi	10	9	7	9	10	9	14	11	1315	601	913	440	1508	1289	4137	2431
ss_pcm	3	3	2	3	3	3	4	3	124	61	102	48	142	131	462	266
systemcaes	11	11	8	13	11	11	16	16	2948	1470	1926	1158	3381	3155	8728	6405
systemcdes	9	8	5	9	9	8	10	10	1125	580	608	351	1290	1245	2755	1942
tv80	17	15	11	15	17	15	22	20	2963	1404	2132	1129	3398	3013	9661	6245
usb_funct	9	9	6	9	9	9	12	10	4906	2227	3408	1775	5626	4781	15443	9816
usb_phy	4	3	2	3	4	3	4	4	188	89	145	60	216	190	657	332
vga_lcd	7	7	6	7	7	7	12	9	40814	18145	28791	16561	46808	38954	130466	91604
wb_conmax	9	7	6	7	9	7	12	10	14580	6966	11003	5895	16721	14954	49860	32608
wb_dma	10	9	6	9	10	9	12	11	1469	711	1129	579	1685	1525	5116	3200
average	7.86	7.14	5.48	7.48	7.86	7.14	10.95	9.14	6406	2985	3711	2142	7346	6408	16816	11849

Table 1: Delay and area comparisons among different architectures

LISP language, and test them on IWLS'05 benchmarks [8]. The MBILP and SAT problem are solved by mosek [24] and zChaff [23], respectively.

Table 1 compares four architectures, i.e., a PLB containing an LUT-4 (column "LUT4"), the mix of an LUT-4 and a macro-gate in Figure 4 (column "LUT4+MG"), an LUT-6 (column "LUT6"), and the mix of an LUT-6 and a macro-gate (column "LUT6+MG"), respectively. The logic depth, number of PLBs, estimated delay and area[7] of the circuits with different architectures are shown in the table. Compared to LUT-4 only architecture, the mix of LUT-4 and macro-gates reduces both logic depth and logic delay by 9.2%, and reduces logic area by 12.9%. Compared to LUT-6 only architecture, the mix of LUT-6 and macro-gates reduces logic delay and logic area by 16.5% and 30%, respectively, while it increases the logic depth by 36.5%. The average logic depth of mixing LUT-6 and macro-gate is larger than that for mixing LUT-4 macro-gates because our technology mapper sets the delay weight of an LUT-6 and a macro-gate as 2.13:1.00 based on Table 2.

In addition, the experimental results show that the functional pruning (FP) technique (Subsection 3.1) used in technology mapping prunes 30% more cuts and reduces 8% overall runtime compared to the algorithm without using FP. More improvement is expected when larger macro-gates are used. Furthermore, the logic area is reduced by 5% due to the MBILP based area recovery algorithm (Subsection 3.2).

6. CONCLUSIONS AND FUTURE WORK

Targeting macro-gate based heterogeneous FPGAs, a methodology has been proposed to extract a small set of logic functions that are able to implement a large portion of functions for given FPGA applications. Assuming that the extracted logic functions are implemented by macro-gates in PLBs, a complete synthesis flow has been developed for such heterogeneous PLBs with mixed LUTs and macro-gates. The flow includes a cut-based delay-optimal technology mapping, a mixed binary integer and linear programming based area recovery algorithm to balance the resource utilization of macro-gates and LUTs for area-efficient packing, and a SAT-based packing. The proposed heterogeneous FPGA design has been evaluated using the newly developed flow. The experimental results show that mixing LUT and macro-gates, both with 6 inputs, improves performance by 16.5% and reduces logic area by 30% compared to using 6-input LUTs.

In the future, we will investigate more complicated macro-gate based heterogeneous PLBs architectures by the proposed synthesis flow, and consider area, delay and power optimization during the technology mapping and packing processes. We will also connect our synthesis flow with the back-end tools such as placement and routing for even more accurate architecture evaluation considering both logic and routing cost.

[7] Given areas in Table 2, the area ratio between these four PLBs is 1.15:(1.15+1):4.53:(4.53+1).

7. REFERENCES

[1] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[2] P. Jamieson and J. Rose, "A verilog RTL synthesis tool for heterogeneous FPGAs," in *FPGA*, 2005.

[3] "The programmable gate array data book," in *Xilinx*, 1989.

[4] J. Cong, H. Huang, and X. Yuan, "Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs," *ACM TODAES*, 2005.

[5] "Xilinx product datasheets," in *http://www.xilinx.com/literature*.

[6] A. Cosoroaba and F. Rivoallon, "Achieving higher system performance with the virtex-5 family of FPGAs," in *http://www.xilinx.com/literature*.

[7] F. Li and L. He, "Power modeling and characteristics of field programmable gate arrays," *IEEE TCAD*, 2005.

[8] "Iwls 2005 benchmarks," in *http://iwls.org/iwls2005/benchmarks.html*.

[9] J. Cong and S. Xu, "Delay-optimal technology mapping for FPGAs with heterogeneous LUTs," in *DAC*, 1998.

[10] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient fpga mapping solution," in *FPGA*, 1999.

[11] A. Kaviani and S. Brown, "Hybrid FPGA architecture," in *FPGA*, 1996.

[12] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," in *IEEE TCAD*, 1994.

[13] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *IEEE TVLSI*, 1994.

[14] D. Chen and J. Cong, "Daomap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *ICCAD*, 2004.

[15] V. Manohara-rajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," in *IWLS*, 2004.

[16] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming," in *IWSL*, 2005.

[17] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor cuts," in *ICCAD*, 2006.

[18] A. Ling, D. Singh, and S. Brown, "Fpga logic synthesis using quantified boolean satisfiability," in *SAT 2005 Springer LNCS Vol 3569*, pp. 444–450.

[19] S. Safarpour, A. Veneris, G. Baeckler, and R. Yuan, "Efficient SATbased boolean matching for fpga technology mapping," in *DAC*

[20] D. Chai and A. Kuehlmann, "Building a better boolean matcher and symmetry detector," in *DATE*, 2006.

[21] S. C. Soheil Ghiasi, Elaheh Bozorgzadeh and M. Sarrafzadeh, "A unified theory of timing budget management," in *ICCAD*, 2004.

[22] "Abc: A system for sequential synthesis and verification," in *http://www.eecs.berkeley.edu/ alanmi/abc/*.

[23] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff:Engineering an endcient SAT solver," in *DAC*, 2001.

[24] http://www.mosek.com.

[25] Jianshe He, "Technology Mapping and Architecture of Heterogenous Field-Programmable Gate Arrays," M.A.Sc. Thesis, University of Toronto, 1993.

[26] Kevin Chung, "Architecture and Synthesis of Field-Programmable Gate Arrays with Hard-wired Connections," Ph.D. Thesis, University of Toronto, 1994.