

LOPASS: A Low-Power Architectural Synthesis System for FPGAs With Interconnect Estimation and Optimization

Deming Chen, *Member, IEEE*, Jason Cong, *Fellow, IEEE*, Yiping Fan, *Member, IEEE*, and Lu Wan, *Student Member, IEEE*

Abstract—In this paper, we present a low-power architectural synthesis system (LOPASS) for field-programmable gate-array (FPGA) designs with interconnect power estimation and optimization. LOPASS includes three major components: 1) a flexible high-level power estimator for FPGAs considering the power consumption of various FPGA logic components and interconnects; 2) a simulated-annealing optimization engine that carries out resource selection and allocation, scheduling, functional unit binding, register binding, and interconnection estimation simultaneously to reduce power effectively; and 3) a *k-cofamily*-based register binding algorithm and an efficient port assignment algorithm that reduce interconnections in the data path through multiplexer optimization. The experimental results show that LOPASS produces promising results on latency optimization compared to an academic high-level synthesis tool *SPARK*. Compared to an early commercial high-level synthesis tool, namely, Synopsys *Behavioral Compiler*, LOPASS is 61.6% better on power consumption and 10.6% better on clock period on average. Compared to a current commercial tool, namely, *Impulse C*, LOPASS is 31.1% better on power reduction with an 11.8% penalty on clock period.

Index Terms—Behavioral synthesis, field-programmable gate array (FPGA), interconnect, power optimization.

I. INTRODUCTION

WITH THE exponential growth of the performance and capacity of integrated circuits, power consumption has become one of the most critical constraining factors in the IC design flow. Rigorous low-power design will require power optimization through the whole design flow to achieve maximal power reduction. A typical VLSI CAD flow goes through multiple design stages, including system-level synthesis, behavioral-level synthesis, register-transfer-level (RTL)

synthesis, logic-level synthesis, and physical design. The higher the design level is, the larger the impact the design decisions impose on the quality of the final product [27].

Our work focuses on power optimization at the behavioral level. Behavioral synthesis (also called high-level synthesis) is a process that takes a given behavioral description of a circuit and produces an RTL design to meet the area, delay, or power constraints. It primarily consists of three subtasks: scheduling, allocation, and binding. Scheduling determines when a computational operation will be executed, allocation determines how many instances of resources (functional units (FUs), registers, and interconnection units) are needed, and binding assigns/binds operations, variables, and data transfers to these resources. The number of resources may be limited, and the total time (latency) to finish the operations can be constrained. These make most of the problems difficult to solve optimally.

Field-programmable gate-array (FPGA) chips are generally perceived as power inefficient because they use a large amount of transistors to provide programmability. Due to the relatively fixed logic and routing resources in a target FPGA platform, it may be difficult to optimize power during the physical design stage, while there are more power reduction opportunities provided by behavioral and architectural design exploration. However, high-level synthesis research specifically targeting FPGA designs for low power is rare. Most previous high-level synthesis techniques for FPGAs optimize objectives other than power reduction. Reference [33] presents a scheduling algorithm for dynamically reconfigurable FPGAs, where FUs can be dynamically reconfigured during runtime to save chip area. In [35], a layout-driven high-level synthesis approach is presented to reduce the gap between metrics predicted during RTL synthesis and the measured data after FPGA implementation. High-level synthesis for a multi-FPGA system is done in [14]. On the low-power part, [34] takes an RTL design and trades off power with circuit speed by selecting different implementations of components iteratively. However, the model presented is quite simplistic and does not consider the power consumption of the steering logic, such as multiplexers (MUXes), and interconnect. Recently, Chen et al. [7] presented a simultaneous resource allocation and binding algorithm for FPGA power minimization, targeting a real FPGA architecture—the Altera Stratix device [2].

In this paper, we will mainly study three interrelated topics for FPGA power reduction: 1) high-level power estimation; 2) simultaneous scheduling, allocation, and binding for power op-

Manuscript received April 28, 2008; revised August 28, 2008 and December 05, 2008. First published June 23, 2009; current version published March 24, 2010. This work was supported in part by Altera Corporation, by the National Science Foundation under Grant CCR-0306682, by the Microelectronics Advanced Research Corporation/Defense Advanced Research Projects Agency Gigascale Systems Research Center, and by Semiconductor Research Corporation-Global Research Collaboration under Grant 2007-HJ-1592.

D. Chen and L. Wan are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA (e-mail: dchen@illinois.edu; luwan2@illinois.edu).

J. Cong is with the Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90095 USA (e-mail: cong@cs.ucla.edu).

Y. Fan is with AutoESL Design Technologies, Inc., Los Angeles, CA 90025 USA (e-mail: fanyp@autoesl.com).

Digital Object Identifier 10.1109/TVLSI.2009.2013353

timization; and 3) interconnection optimization. Correspondingly, we build a high-level power estimator and an interconnect-centric power optimization engine into our high-level architectural synthesis system, i.e., LOPASS.

High-Level Power Estimation: To perform effective power optimization in behavioral synthesis, we need to estimate power consumption at a higher level of abstraction before the low-level details of the circuit are finalized. An accurate and efficient power estimator will provide invaluable directions for the optimization process. As technology scales, interconnect power consumption dominates the overall submicrometer FPGA power consumption. For example, more than 80% of the total power is consumed in interconnects (including clock networks) in 0.1- μm technology for certain FPGA architectures [23], [24]. Similar results were reported in other research, such as [21] and [28]. Consequently, power estimation in behavioral synthesis must consider the total wire capacitance. We develop a high-level power estimator that takes into account special FPGA architecture characteristics, the associated CAD design flow, and the estimated total wire length in the FPGA design. This power estimator is flexible and can support a variety of FPGA architecture parameters. We use a fast switching activity calculation algorithm [4] and enhance it to improve its flexibility for handling different scheduling, allocation, and binding situations.

Power Optimization Engine: Since the subtasks of behavioral synthesis are highly interrelated and the objectives may conflict with each other, sequentially optimizing them one by one may not produce satisfactory solutions. The goal of our power optimization is to search a combined solution space for the subtasks in behavioral synthesis so that we can not only minimize the power of FPGA designs but also meet the performance/latency targets. To achieve this goal, we adopt a simulated-annealing-based algorithm. Simulated annealing has been applied in previous research to carry out scheduling, allocation, and binding subtasks concurrently to reduce hardware area [11], [20]. The key difference in our simulated-annealing engine is that we generate the full data path after each move and capture the overall cost, considering all the contributing factors in the design.

Interconnection Optimization: The task of connecting the FUs and registers together is called interconnection (or simply connection) allocation. A MUX is a standard complex gate that is often used in data-path logic to multiplex signals between FUs and registers. The total interconnections in the design determine the total MUX inputs or MUX connectivity. The objective of interconnection allocation is to reduce the MUX requirement. Register binding offers good opportunities for interconnection sharing because it influences how the data are stored and transferred on the interconnections. In addition, assigning interconnections to different input ports on FUs can also influence the MUX sizes. Therefore, we work on register binding and port assignment for MUX reduction. The experimental results show that our solution provides significant improvement compared to a previously published algorithm for MUX reduction.

In the following, Section II introduces a generic FPGA architecture and our overall LOPASS synthesis flow. Section III describes the power estimator. Section IV describes the optimiza-

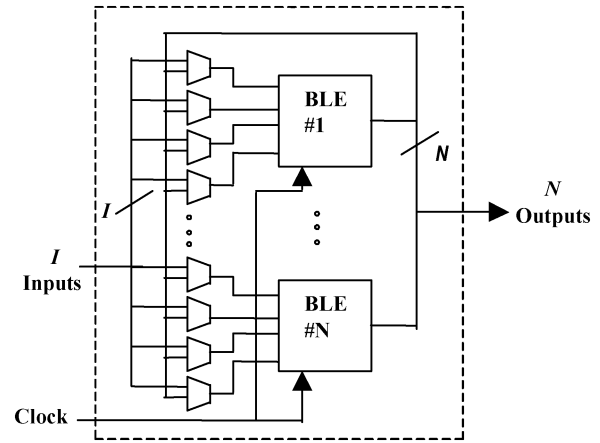


Fig. 1. CLB.

tion engine, and Section V describes the multiplexer optimization algorithm. Section VI discusses the experimental results, and we conclude this paper in Section VII.

II. FPGA ARCHITECTURE AND OVERALL SYNTHESIS FLOW

A. FPGA Architecture

The dominant FPGA technology today is based on static random-access memories (SRAMs), in which programmability is realized using SRAM cells to implement programmable logic elements (LEs), programmable I/Os, and routing elements. The common approach for implementing a basic LE (BLE) is to use a K -input one-output lookup table (K LUT) of 2^K SRAM cells. A K LUT can implement any Boolean functions of up to K variables by loading the SRAM cells with the truth table of that function. A group of LUTs can form a configurable logic block (CLB), as shown in Fig. 1. A size- N logic block contains N BLEs. The block inputs and outputs are fully connected to the inputs of each LUT.

The interconnect structure can be modeled as 2-D segmented wire channels connected by programmable switch boxes. The island-style FPGA, the most popular FPGA architecture, surrounds each CLB with interconnects on the four sides of the CLB and builds the connections in between through programmable connection boxes. Fig. 2 shows a high-level view of an island-style FPGA, which is the architecture model adopted by a popular academic tool VPR [3].

By varying the architecture parameters for logic blocks (K and N) and routing resources, one can easily create many different FPGA architectures. One key parameter for routing is the wire segment length, which is defined as the number of logic blocks that each wire segment spans. Wire segments are connected by routing switches in the routing channels. Programmable routing switches are either pass transistors or tristate buffers. There are also switches (in the connection boxes) connecting the wire segments to the logic block inputs and outputs (Fig. 2). In [3], routing architectures are defined by the parameters of channel width (W), switch box flexibility (F_s —the number of wires to which each incoming wire can connect in a switch box), connection box flexibility (F_c —the number of wires in each channel to which a logic block input

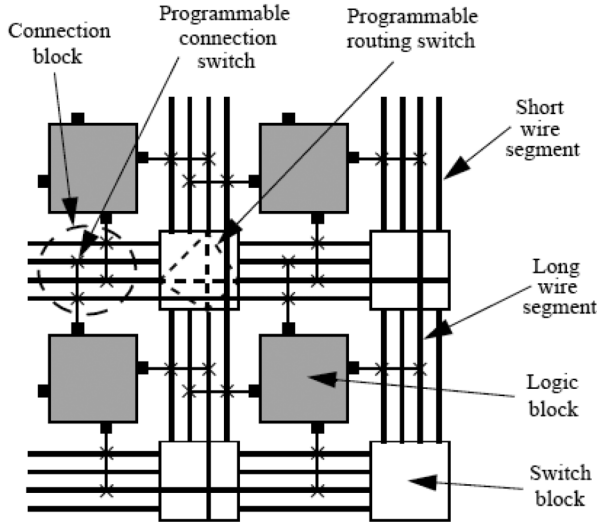


Fig. 2. Island-style FPGA [3].

or output pin can connect), and segmented wire lengths. In this paper, we will use logic block size N as 4 and LUT input size K as 4. All the wire segments are length-1 segments, and all the routing switches are tristate buffers. This architecture is similar to the one used in [29]. We believe that our optimization results hold for similar architectures with different logic or routing parameters. We use 0.1- μm technology in this paper, which is the same as that used in [23] and [24].

B. Overall Synthesis Flow

Fig. 3 shows our overall LOPASS synthesis flow. A design in a high-level description language is converted into a control–data flow graph (CDFG) or DFG as its internal representation. Our low-power optimization engine then takes the latency or resource constraints from the user and starts the power optimization process through simulated annealing. For the explored FU allocation and binding solutions that fulfill the latency constraint, we examine their impact on the resource usage, steering logic, and interconnections. All of these data are used by the power estimator, which then feeds the estimated power value back to the power optimization engine to guide the simulated-annealing process. Such a power optimization already considers interconnection reduction. In addition, after the power optimization engine exits, a more sophisticated postprocessing algorithm will be conducted for further multiplexer and interconnection reduction. We then generate the RTL design and use an RTL synthesis tool, namely, *Design Compiler*, from Synopsys [32] to map the design into a gate-level netlist. Afterward, we are able to report the delay, power, and area values of the design by a gate-level FPGA evaluation tool *fpgaEva_LP2*[24]. Notice that the high-level power estimator uses the same FPGA architecture parameters as *fpgaEva_LP2* does. Since LOPASS closely reflects the modeled FPGA architecture, it can be used to evaluate different FPGA architectures for their power efficiency from a high-level design perspective. The gray areas in Fig. 3 are the focus of this paper. Other steps in the flow will also be explained briefly later in this paper.

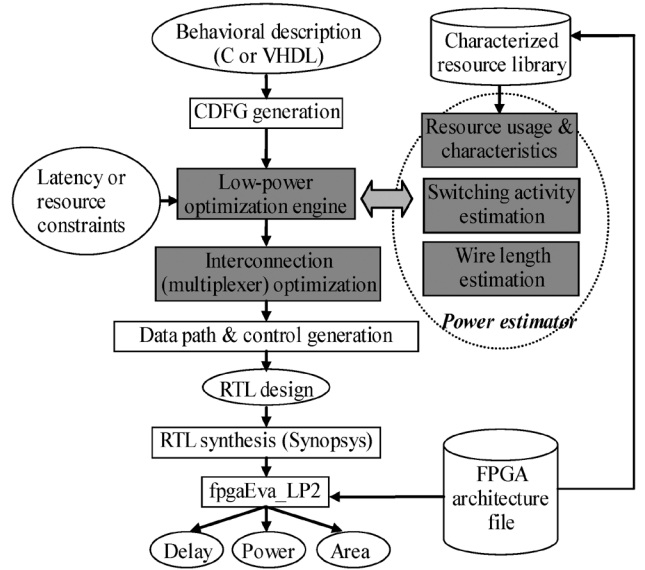


Fig. 3. LOPASS synthesis flow.

III. FPGA HIGH-LEVEL POWER ESTIMATION

We will first introduce three major components for our high-level FPGA power estimation: wire length estimation, switching activity estimation, and resource library characterization. We then present how these components are weaved together to form the estimator.

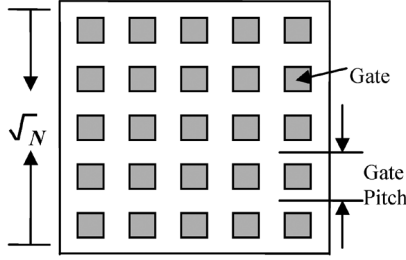
A. Wire Length Estimation

Wire length estimation before layout has been one of the most important applications of *Rent's rule*. Rent's rule was first introduced in 1960 by E. F. Rent of IBM, who published an internal memorandum for log plots of “number of pins” versus “number of circuits” in a logic design. Such plots tend to form straight lines in a log–log scale and follow the following relationship:

$$T = kN^p \quad (1)$$

where T is the number of external pins of a logic network, N is the number of gates contained in the network, k is the average number of pins per gate in the network, and p is Rent's exponent. A series of works followed. Particularly, the work [9] offers a complete description of wires of different lengths for targeted microprocessor architectures. It models the architecture as homogeneous arrays of gates evenly distributed in a square die. This architecture model reflects the characteristics of an island-style FPGA if we treat each logic block (CLB) of the FPGA as a multioutput gate in the model.

Fig. 4 shows the architecture model with N gates [9] (note that this N is different from the N in Fig. 1). The length of one side of the square chip is \sqrt{N} in units of gate pitches. We will apply the interconnect density function derived in [9]. Equation (2) shows the details, where $i(l)$ is the interconnect density function. $I(a < l < b)$ gives the total number of interconnects between lengths $l = a$ and $l = b$ (l is in units of logic block pitches for our FPGA). In the equation, N is the number of CLBs in the design, p is Rent's exponent, α is the fraction of the on-chip terminals that are sink terminals, $f.o.$ is the average fan-out,

Fig. 4. Targeted architecture with N gates.

k is the average number of input/output pins per CLB, and Γ represents a constant calculated using N and p (see the details in [9]). Notice that $i(l)$ is calculated by different formulas for wire lengths between 1 and \sqrt{N} , and \sqrt{N} and $2\sqrt{N}$, respectively. For example, to compute the number of length-5 wires in routing channels (signals travel across five logic blocks to reach their destinations), we use $I(4 < l < 5)$. After obtaining the numbers of wires for all the lengths (from 1 up to $2\sqrt{N}$), we sum all the wire lengths together as the total wire length estimation. We use a Rent's exponent extracted from [29] because it explores a similar FPGA architecture, and the placement and routing flow is quite similar. This is important because p is an empirical constant that closely relates to the architecture and design flow. The proposed wire estimation model is very general for total wire length estimation as long as Rent's exponent is derived accurately. In our paper, a typical value of k is 10.5, a typical value of $f.o.$ is 1.8, and p is 0.62. The experimental results show that this wire estimation model is valid, and its prediction is reasonably accurate compared to the final total wire length after placement and routing.

$$\begin{aligned}
 \text{Region I: } & 1 \leq l < \sqrt{N} \\
 & i(l) = \frac{\alpha k}{2} \Gamma \left(\frac{l^3}{3} - 2\sqrt{N}l^2 + 2Nl \right) l^{2p-4} \\
 \text{Region II: } & \sqrt{N} \leq l < 2\sqrt{N} \\
 & i(l) = \frac{\alpha k}{6} \Gamma (2\sqrt{N} - l)^3 l^{2p-4} \\
 \text{where } & \alpha = \frac{f.o.}{f.o. + 1} \\
 \text{such that } & I(a < l < b) = \int_a^b i(l) dl. \quad (2)
 \end{aligned}$$

Note that this total wire length estimation is for global wires used in the routing channels, which does not include the wires used within CLBs. We call the wires within CLBs local wires, which can be easily estimated by the size of CLBs using the CLB architecture shown in Fig. 1.

B. Switching Activity Estimation

This section discusses an efficient switching activity calculator using CDFG simulation, extending the method from [4] that performs simulation just once at the initialization and afterward computes switching activities for any legal binding without repeating simulations.

Given a scheduled and bound CDFG G , every node (or operation) of G is bound to an FU and scheduled to a certain control step. Many operations may share a common FU. In other words, an FU will execute a sequence of operations in a fixed order after binding and scheduling. For an FU U , we define $C_{\text{in}}(O, O')$ as the *toggle count* from executing operation O to operation O' . This number represents the input transitions when the FU switches the execution from O to O' . Notice that U has two ports. For simplicity, we use $C_{\text{in}}(O, O')$ to represent the input toggle counts of both ports. Let $(PI^1 \rightarrow PI^2 \rightarrow \dots \rightarrow PI^K)$ be a sequence of stimuli enforced on the primary inputs of G . By performing functional simulation on G , with primary input stimulus PI^j ($1 \leq j \leq K$), we can obtain input bit vector I_i^j for operation O_i ($1 \leq i \leq N$). I_i^j is computed based on the propagation of PI^j through the design when the propagation reaches the internal operational node O_i . For an FU U , let $(O_1 \rightarrow O_2 \rightarrow \dots \rightarrow O_N)$ be the *operation sequence* in the execution order (O_1 to O_N are bound to U). $C_{\text{in}}(O_i, O_{i+1})$, for $1 \leq i < N$, and $C_{\text{in}}(O_N, O_1)$, under this primary-input stimulus sequence, are then defined as follows [4]:

$$C_{\text{in}}(O_i, O_{i+1}) = \sum_{j=1}^K D_{\text{H}} \left(I_i^j, I_{i+1}^j \right) \quad (3)$$

$$C_{\text{in}}(O_N, O_1) = \sum_{j=1}^{K-1} D_{\text{H}} \left(I_N^j, I_1^{j+1} \right) \quad (4)$$

where $1 \leq i < N$, and $D_{\text{H}}(X, Y)$ represents the Hamming distance between bit vectors X and Y . Notice that $C_{\text{in}}(O_N, O_1)$ represents the toggle count between O_N and O_1 when the execution finishes O_N and begins a new iteration starting with O_1 with the bit vector I_1^{j+1} .

The *switching activity* S_{in} of the inputs of an FU U is the ratio of the number of bit flips observed on its inputs between cycles over the maximum possible number of bit flips. S_{in} is formally defined as

$$S_{\text{in}} = \frac{\sum_{i=1}^{N-1} C_{\text{in}}(O_i, O_{i+1}) + C_{\text{in}}(O_N, O_1)}{2 \times \text{Bit_width} \times (N \times K - 1)} \quad (5)$$

where Bit_width is the input vector width of U .

In [4], a matrix of C_{in} is constructed after scheduling but before binding and is used to calculate S_{in} for every possible binding solution. Two operations are compatible if they can be bound to the same FU. For two compatible operations O_i and O_j , there will be two entries $[O_i, O_j]$ and $[O_j, O_i]$ in the pre-calculated matrix. Supposing that O_i is scheduled before O_j , the value of matrix element $[O_i, O_j]$ is calculated with (3). The value of $[O_j, O_i]$ is calculated from (4). After binding, the operation sequence is known for every FU, and every C_{in} value is looked up in the matrix. The input switching activity can be calculated accordingly based on (5). The output switching activity can be computed following a similar approach using a C_{out} matrix [4]. Scheduling cannot be changed after the $C_{\text{in}}/C_{\text{out}}$ matrices are constructed in [4].

To make switching activity estimation more flexible, we extend the C_{in} matrix to support every possible scheduling and binding situation. That is, for every two compatible operations

O_i and O_j , we precalculate the C_{in} values for both scheduling orders ($O_i \rightarrow O_j$) and ($O_j \rightarrow O_i$), using (3) and (4), so that there will be two values for each scheduling order. With this enhanced matrix, regardless of how O_i and O_j are scheduled and bound, we can find the entries in the matrix when calculating S_{in} . For the output-signal toggle count C_{out} and the output switching activity S_{out} , we use a similar method. An FU is a subcircuit. Because we do not have the detailed information of the internal structure of the FU, we do not have switching activities of the internal gates of the FU. An intuitive way to estimate the switching activity of an FU U is to use $S_U = (S_{in} + S_{out})/2$.

The total switching activity of a scheduled and bound CDFG, i.e., S_{design} , is the weighted average of switching activities for all used FUs. We use a larger weight for multipliers because the switching power of a multiplier is much larger than that of an adder per switch. Therefore, the switching activities of the multipliers are given more weight for the overall design. Equation (6) shows the details

$$S_{design} = \frac{\sum_{i=1}^Q S_i + \gamma \cdot \sum_{j=1}^M S_j}{Q + \gamma \cdot M} \quad (6)$$

where Q is the number of adders/subtractors and M is the number of multipliers used in the design (the set of benchmarks that we use only contain addition/subtraction and multiplication operations, while the equation can be easily extended to support more types of FUs). γ is the weight, which represents a typical ratio of the dynamic power of a multiplier over the dynamic power of an adder based on resource characterization (more details in the next section).

C. Resource Library Characterization

We use a resource library that is derived from the DesignWare libraries available from Synopsys [32]. Therefore, our characterization for the resources closely correlates to the characteristics of real-life implementations of the resources. We provide different resource versions for implementing the same operation type to offer larger design flexibility. These resources have different area, delay, and power characteristics. Given different user requirements, the behavioral synthesis should be able to find a good tradeoff between latency and power, where latency is defined as the product of the *clock period* and the *total number of clock cycles* used for the design.

Under this assumption, we select adders, multipliers, and multiplexers and characterize their area, delay, and power. Fig. 5 shows the flow for this characterization. A DesignWare IP component goes through Synopsys *Design Compiler* for synthesis and mapping. We use a generic *lsi_10k* library available in the Synopsys tool. Design Compiler transforms the IP core into a gate-level netlist consisting of only two-input gates and flip-flops. The netlist is in a structured VHDL format. We then convert the design into the Berkeley logic interchange format (BLIF). Then, *fpgaEva_LP2* takes the gate-level BLIF design, goes through synthesis and physical design stages, and reports the delay, area, and power values of the component.

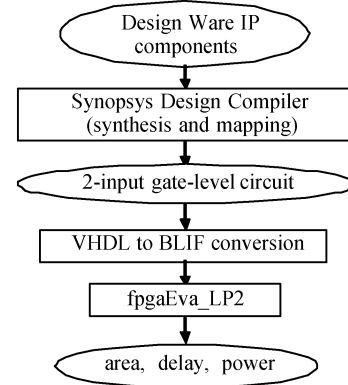


Fig. 5. Resource characterization flow.

TABLE I
CHARACTERIZATION DATA WITH FPGA IMPLEMENTATION (0.1- μ m TECHNOLOGY)

Resources	Implementation	Area (CLB)	Delay (ns)	Power (W)
add24b_bk	Brent-Kung	42	6.09	0.016
add24b_cla	Carry look-ahead	26	11.78	0.010
mul18b_wall	Booth-recoded Wallace tree	280	14.78	0.308
mul18b_wall_s2	Wallace tree	286	11.43	0.164
mul18b_wall_s4	Wallace tree	262	7.14	0.114
mux24b_2to1	Synopsys synthesis	6	0.57	0.002
mux24b_4to1	Synopsys synthesis	18	2.34	0.005
mux24b_8to1	Synopsys synthesis	66	4.60	0.023
mux24b_16to1	Synopsys synthesis	135	6.91	0.083
mux24b_32to1	Synopsys synthesis	276	10.93	0.240

Table I shows some of the characterization data. The *area* (in terms of number of CLBs) required for the FPGA implementation of the resource, the critical path *delay* after placement and routing, and the total *power* values are reported. The average number of input/output signals per CLB and the average gate-fan-out number of resources are also recorded because they are used in the calculation of the wire distributions (Section III-A). The delay values would determine the clock period of the design. Among all the resources used for a design, the minimum *delay* value determines the clock period, and other resources with larger *delay* values will run in a multicycle mode. For example, if a design only uses carry look-ahead adders and Booth-recoded Wallace tree multipliers, the clock period of the design will be set to 11.78 ns. As a result, the adder takes one clock cycle to finish its execution, while the multiplier takes two clock cycles to finish its execution.

In our benchmarks, the output of a multiplier never exceeds 24 b, and the input of a multiplier never exceeds 18 b. Therefore, we use 18-b multipliers and 24-b adders. The *power* values can be used to estimate the weight γ in (6). However, they are not directly used in our power estimator because they only represent the atomic power values of individual resources by themselves. To have an accurate power estimation for the whole design, our power model considers detailed power characterization for LEs used by the entire design in both FUs and interconnection units

(MUXes). We also estimate the power of global wires incurred for the connections of these elements. Details will be presented in the next section.

D. Put It All Together: High-Level Power Estimator

We consider both *dynamic* and *static* powers for various FPGA components in our power model. FPGAs contain buffer-shielded LUT cells with fixed capacitance load. Therefore, we can use precharacterization-based macromodeling to capture the average switching power per access of an LUT and register. As for interconnects, a switch-level calculation can be used because their capacitance cannot be predetermined. This mixed-level FPGA power model is similar to that used in the gate-level power model in [23] and [24]. The difference is that we estimate power at a higher level, which is much faster than a gate-level power estimator. Since the high-level power estimator is used to guide the power optimization engine, it has to be called by the power engine very frequently. Therefore, a fast power estimator with sufficient accuracy is helpful for the overall power optimization.

Our high-level power model can be summarized in (7) and (8). The dynamic power is contributed from P_{LUT} (for all the LUTs contained in the design), P_{REG} (for all the registers), P_{LW} (for local wires within the CLB), and P_{GW} (for global routing wires)

$$P_{Dynamic} = P_{LUT} + P_{REG} + P_{LW} + P_{GW} \quad (7)$$

$$P_{Static} = P_{Static_LUT} + P_{Static_FF} + P_{Static_LB} + P_{Static_GB}. \quad (8)$$

P_{LUT} is calculated through $N_{LUT} \cdot S \cdot E_{LUT} \cdot f$, where N_{LUT} is the total number of LUTs in the design; S is the estimated switching activity of the design (Section III-B); E_{LUT} is the energy consumption per switching of the LUT [23], [24]; and f is the frequency. P_{REG} is calculated similarly through $N_{REG} \cdot S \cdot E_{REG} \cdot f$. For the wires, P_{LW} and P_{GW} are calculated through the formula $0.5f \cdot S \cdot V_{dd}^2 \cdot C_{Wire}$, where f and S are the same as before, V_{dd} is the supply voltage, and C_{Wire} is the estimated wire capacitance for either local wires within the logic block or global wires in the routing tracks. The local wires in a CLB can be estimated through the CLB architecture shown in Fig. 1 (see [23] and [24] for a detailed CLB structure). The capacitance of local buffers (LBs) and local multiplexers is lump-summed into the local C_{Wire} . The global wire length estimation is carried out using the method in Section III-A. The capacitance of buffers and/or pass transistors (in switch boxes and connection boxes) in global interconnects is lump-summed into the global C_{Wire} according to the routing architecture.

In (8), the static powers of all the used LUTs, FFs, LBs, and global buffers (GBs) are included. The static powers of these circuit components are borrowed from [23] and [24], where the authors obtained static-power macromodels based on SPICE simulation. We also calculate the numbers of idle components through our architecture model and the estimated chip size of the FPGA and count their static powers as well.

Since we only perform initial circuit simulation once, and the switching activities for different binding and scheduling scenarios can be estimated by simple lookups in the C_{in} and

C_{out} matrices, our high-level power estimator runs extremely fast—almost close to constant time, which is the key needed to speed up the power optimization engine.

IV. POWER OPTIMIZATION ENGINE

Before we introduce the power optimization engine, we will examine in the following some of the FPGA's unique features that will help us gain some insights for designing efficient optimization algorithms:

- 1) FPGAs offer an abundance of distributed registers;
- 2) FPGAs have no efficient support for wide MUXes (Table I);
- 3) smaller numbers of FUs and/or registers may not correspond to a smaller area or power.

These facts will influence register allocation and binding and steering logic allocation during behavioral synthesis. Particularly, since FPGAs are not efficient in implementing wide-input MUXes due to limited routing resources, solely working for a smaller number of FUs may lead to unfavorable solutions because it may generate a larger number of wide-input MUXes that outweigh the gain on FU reduction. Therefore, we need an algorithm that explores a large solution space considering multiple constraining parameters that influence FU and register allocation, binding, MUX generation, and scheduling.

As mentioned in the Introduction, we adopt the simulated-annealing process as our power optimization engine. Its basic feature is to allow hill-climbing moves to explore the solution space. The probability of accepting these moves is controlled by a parameter that is analogous to temperature in the physical annealing process. Temperature decreases gradually as the annealing process proceeds. We start our annealing process with an initial FU allocation and binding solution generated by a simple greedy algorithm under the latency constraint. The engine then performs five types of binding moves to gradually reduce the overall cost. The cost is the total power consumption calculated by the high-level power estimator. The moves are randomly picked, and the targeted FU(s) for each move is randomly picked as well. The moves are as follows.

- 1) **Reselect**: Select another FU of the same functionality but with a different implementation. For example, select a carry look-ahead adder to replace a Brent–Kung adder. The operations that are bound to the adder stay unchanged.
- 2) **Swap**: Swap two bindings of the same functionality but different implementations. It is equivalent to two *reselects* between two FUs in each direction.
- 3) **Merge**: Merge two bindings into one, i.e., the operations that are bound to two FUs are combined into one FU. As a result, the total number of FUs decreases by one.
- 4) **Split**: Split one binding into two. It is the reverse action of *merge*. As a result, the total number of FUs increases by one. The operations of the original binding are distributed into the new bindings randomly.
- 5) **Mix**: Select two bindings, *merge* them, sort the merged operations according to the decreasing order of their timing slacks, and then *split* the operations. For example, if there are N operations after sorting, operations 1 to $N/2$ will form one binding with one FU, and the rest of the operations will form another binding.

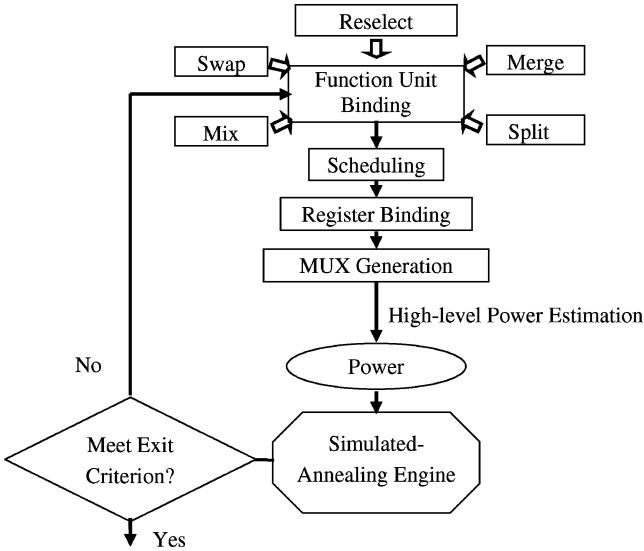


Fig. 6. Power optimization engine.

Each of these moves has its own attributes. *Reselect* may pick a smaller FU (possibly larger delay) to reduce power if the latency constraint can still be met. *Mix* may lead to rebinding the operations that have larger slacks into a pipelined FU such as *mul18 bit_wall_s4* (Table I), which consumes much less power but has a longer latency (four cycles to finish the operation). *Split* will be disabled when temperature is low, so the allocation and binding solution will not be dramatically changed. After each move, resource-constrained list scheduling [10] is called to verify the total latency. If latency is violated, the move is discarded. To efficiently estimate the steering logic for the current binding in minimal runtime, a left-edge algorithm [10] is used for register binding. Next, FUs and registers are connected together through multiplexers. The characteristic data of the current move are fed into our power estimator to estimate the cost. The annealing process exits when temperature is low enough.

Fig. 6 shows the overall flow of the optimization engine in a block diagram. After the simulated-annealing-based power optimization exits, we redo register binding for the purpose of minimizing the total multiplexer usage. A port assignment algorithm further reduces the sizes of multiplexers. These two optimization procedures are not included in the annealing schedule because they take significant runtime and may make the overall runtime of the annealing schedule intolerable. We will present these multiplexer optimization algorithms in their entirety in the next section.

V. MULTIPLEXER OPTIMIZATION FOR INTERCONNECTION REDUCTION

In this section, we present a-cofamily-based algorithm to carry out the register binding task, which guarantees a minimum number of registers while reducing the MUX usage. We also implement a port assignment algorithm that further reduces the total MUX inputs efficiently after register binding. Next, we will first introduce some related definitions and the problem formulations. Then, we present our register binding and port assignment algorithms separately.

A. Definitions and Problem Formulation

Given a DFG, $G = (V, A)$, let $V = \{v_1, v_2, \dots, v_x\}$ and $A = \{a_1, a_2, \dots, a_y\}$, and $a = \{v_m, v_n\}$ represents the edge from v_m to v_n . Node set V corresponds to operations, and edge set A corresponds to data flowing from one operation to another. After scheduling, the lifetime of each data value (or variable) in the DFG is the time during which the data value is active (valid) and can be defined by an interval [birth time; death time]. *Birth time* is the control step when the variable becomes available. For example, if an addition A_1 can finish execution in one clock cycle, and it is scheduled to control step d , the birth time of the variable generated by A_1 will be $d + 1$. *Death time* is the control step when the variable is last used (the variable can flow to multiple places through different data edges). A variable stays in a single register during its entire lifetime until it is replaced by another variable. A compatibility graph $G_c = (V_c, A_c)$ for these variables can then be constructed, where vertices correspond to variables, and there is a directed edge $a_c = (v_i, v_j)$ between two vertices if and only if their corresponding lifetimes do not overlap, and variable v_i comes before v_j . In such a case, we call variables v_i and v_j compatible with each other, and they can be bound into a single register without lifetime conflicts. Let w_{ij} denote the weight of edge a_c , which represents the cost of binding v_i and v_j into a single register. This cost will be the estimated interconnection cost in our case.

The initial design input comes out of the power optimization engine, where FU allocation/binding/scheduling operations are all done. Register allocation and binding are also done. We change this initial solution with two additional optimization steps: redoing register binding and adding port assignment for the purpose of reducing MUX inputs. We first formally define the register binding and port assignment problems. Both problems are NP hard [26].

Register Binding for MUX Reduction: Instance: A scheduled DFG $G = (V, A)$, a set of registers R , a set of FUs F , an FU binding $\{f_u : v \rightarrow u\}$ for all v , where $v \in V$ and $u \in F$, and a positive integer N .

Question: Is there a register binding $\{f_r : a \rightarrow r\}$ for every variable a , where $a \in A$ and $r \in R$ such that the number of interconnections between registers and FUs is not larger than N ?

Port Assignment for MUX Reduction: Instance: A scheduled DFG $G = (V, A)$, a set of registers R , a set of FUs F , an FU binding $\{f_u : v \rightarrow u\}$ for every operation v , where $v \in V$ and $u \in F$, a register binding $\{f_r : a \rightarrow r\}$ for every variable a , where $a \in A$ and $r \in R$, and a positive integer N .

Question: Is there a port assignment, i.e., for every operation that is bound to an FU u , considering the two respective input registers containing the two operands for the operation, whose input register should be connected to which port of u , such that the number of total interconnections between all the input registers (for all the operations that are bound to u) and u is not larger than N ?

B. Register Binding With k -Cofamily Formulation

1) Problem Reduction: We state our register binding objective as follows. Given a compatibility graph $G_c = (V_c, A_c)$, find

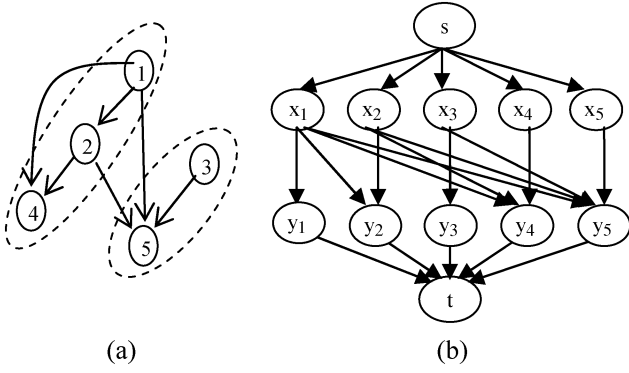


Fig. 7. POSET and its split graph. (a) POSET P_c . (b) Split graph $G(P_c)$.

a subset of A_c that covers all the vertices in V_c in such a way that the total sum of the weights of all the edges in the subset is the minimum, with the constraint that all the vertices can only be bound into as many as k registers as possible. If we find such a subset, we say that we carry out register binding of V_c into k registers with the *minimum total weight*.

We formulate our register binding problem as a problem of calculating the minimum weighted *cofamilies* of a *partially ordered set (POSET)*. A POSET P is a collection of elements with a binary relation \leftarrow defined on $P \times P$ that satisfies the following conditions [25]:

- 1) *reflexive*, i.e., $x \leftarrow x$ for all $x \in P$;
- 2) *antisymmetric*, i.e., $x \leftarrow y$ and $y \leftarrow x \Rightarrow x = y$;
- 3) *transitive*, i.e., $x \leftarrow y$ and $y \leftarrow z \Rightarrow x \leftarrow z$.

We say that x and y are *related* if we have either $x \leftarrow y$ or $y \leftarrow x$. An *antichain* in P is a subset of elements such that no two of them are related. A *chain* in P is a subset of elements such that every two of them are related. A k -*family* in P is a subset of elements that contains no chain of size $k + 1$, and a k -*cofamily* in P is a subset of elements that contains no antichain of size $k + 1$ [16]. We show an example in Fig. 7. In Fig. 7(a), edges represent *relations* among the elements. The POSET contains a 3-*family* and a 2-*cofamily*. We can associate weights for k -cofamilies, where the minimum weighted k -cofamilies are particularly important to us. We now build the relationship between a compatibility graph and a POSET.

Given a compatibility graph $G_c = (V_c, A_c)$, let POSET $P_c = \{v_1, v_2, \dots, v_n\}$ such that P_c contains all the vertices of G_c , and the compatibility relation defined in A_c can be the relation \leftarrow on the elements of P_c . It is easy to show that the compatibility relation is reflexive, antisymmetric, and transitive. By such, an edge $a_c = (v_i, v_j)$ of A_c represents a relation between the two elements of P_c as $v_i \leftarrow v_j$. Therefore, there is a one-to-one correspondence between one node in V_c and one element in P_c , and between one edge in A_c and one \leftarrow in P_c . We also assign the weight on a_c to the relation $v_i \leftarrow v_j$. There is no weight on the element itself. We have the following result.

Theorem 1: Register binding on a compatibility graph G_c into k registers is equivalent to finding k disjoint chains in the POSET P_c , and each chain contains all the variables that are bound into one of the k registers.

Proof: A register binding solution gives a grouping of variables, and each group is assigned (bound) to a different register. All the variables assigned to a register are compatible with

one another, and their lifetimes are sequential and not overlapping. Therefore, the elements corresponding to the variables in a group form one chain in the POSET. Since there is no variable that is assigned to more than one register, the grouping determines a disjoint chain in the POSET. The direction from k disjoint chains to k groups of variables holds true as well. ■

Theorem 1 can be shown in Fig. 7(a). The solution with an optimal number of registers (in this case, two) is obtained by the partition of two disjoint chains (dashed ovals) in the POSET. Variables in one chain can then be bound into one separate register.

Let t_i represent the weight of a chain C_i in a POSET. Suppose that $C_i = \{v_1, v_2, v_3, \dots, v_{x-1}, v_x\}$, $t_i = \sum_{k=1}^{x-1} w_{k,(k+1)}$, where $w_{k,(k+1)}$ is the weight assigned to the relation $v_k \leftarrow v_{k+1}$. Register binding of the nodes in G_c into k registers with the minimum total weight is equivalent to finding k disjoint chains in the POSET P_c with the minimum total weight from the k chains. There is an important fundamental result on POSETs due to Dilworth [12], which indicates that any k -cofamily in a POSET P can be partitioned into at most k disjoint chains. A chain is *nontrivial* if it contains at least two elements. If an element v_i is not related to any other elements in the POSET, we say that v_i forms a *trivial* chain just by itself. We say that a k -cofamily is *nontrivial* if it can be partitioned into exactly k disjoint *nontrivial* chains. We have the following corollary.

Corollary 1: The minimum weighted nontrivial k -cofamily with at least one antichain of size k in a POSET P_c can be partitioned into exactly k disjoint nontrivial chains with the minimum total weight to cover every element in P_c , when the weights on the relations are all negative values.

Proof: It is a simple extension based on the Dilworth theorem. Any k -cofamily with at least one antichain of size k can be partitioned into k disjoint chains. In other words, the k disjoint chains form a k -cofamily for the POSET. If the k disjoint chains hold the minimum total weight, then the k -cofamily holds the minimum total weight as well. The minimum weighted k -cofamily will cover all the elements in P_c . If it is not the case, adding more elements into the k -cofamily will always reduce the total weight. ■

Therefore, our goal is to find the minimum weighted k -cofamily in P_c (or the minimum weighted k disjoint chains in P_c). In [8], an algorithm based on network flow theories was presented to calculate the maximum node-weighted k -cofamilies (weights on nodes only). In this paper, we show how to convert the calculation of the minimum relation-weighted k -cofamily into the calculation of the minimum-cost flow in a network.

First, we construct our network flow graph, the split graph $G(P_c)$, as follows. For each element v_i in P_c , we introduce two vertices x_i and y_i in $G(P_c)$ and a directed edge (x_i, y_i) . We introduce a directed edge (x_i, y_j) in $G(P_c)$ if $v_i \leftarrow v_j$ ($i \neq j$). Moreover, we introduce two more vertices s (source) and t (sink) in $G(P_c)$ and add edges (s, x_i) and (y_i, t) for each $1 \leq i \leq n$. Fig. 7(b) shows the corresponding split graph of POSET P_c of Fig. 7(a). We set the capacity of each edge e to be one and the cost of each edge e , denoted as $d(e)$, to be

$$d(e) = \begin{cases} 0, & \text{if } e = (s, x_i) \text{ or } (y_i, t) \\ w_{ij}, & \text{if } e = (x_i, y_j) \text{ (} i \neq j \text{)} \\ 0, & \text{if } e = (x_i, y_i) \end{cases} \quad (9)$$

where w_{ij} is a negative value (to be covered in the next section). Therefore, the cost of the flows in $G(P_c)$ will always be smaller by going through edges (x_i, y_j) ($i \neq j$) instead of edges (x_i, y_i) . We have the following theorem.

Theorem 2: Let P_c be a POSET of n elements, and the largest antichain in P_c has size k . Then, P_c has a k -cofamily that includes all n elements with the minimum total weight if and only if the split graph $G(P_c)$ has an $(n-k)$ -flow of the minimum total weight.

Proof: We will first show that the theorem holds when P_c has a nontrivial k -cofamily, i.e., it can be partitioned into k nontrivial chains. We then extend our result to k -cofamilies that contain trivial chains.

(If) Suppose that $G(P_c)$ has a $(n-k)$ -flow of minimum total weight (minimum-cost $(n-k)$ -flow) and that there are no flows that pass through edges (x_i, y_i) . Therefore, there are $n-k$ numbers of flows, and each of them passes a separate edge like (x_i, y_j) ($i \neq j$) since all the edge capacities are one. These flows will form $n-k$ disjoint paths in $G(P_c)$ (except the starting node s and the ending node t). By Corollary 1, the minimum-weighted nontrivial k -cofamily can be partitioned into k disjoint nontrivial chains with the minimum weight. The chains can be formed as follows. We start with n elements in P_c , i.e., n single-element chains. For each edge (x_i, y_j) ($i \neq j$) with a unit flow in $G(P_c)$, we join element i and element j together, which reduces the total number of chains by one. Along this process, the chains formed are all disjoint with one another. We picked $n-k$ number of such edges, so the number of chains left is eventually $n - (n-k) = k$. By such, disjoint k chains are formed, and each chain is nontrivial. Since the $(n-k)$ -flow has the minimum total weight, the k -cofamily thus formed has the minimum total weight as well.

(Only If) Suppose that the k disjoint nontrivial chains are S_1, S_2, \dots, S_k . Let β_i denote the size of S_i ($1 \leq i \leq k$). Then, $n = \sum_{i=1}^k \beta_i$. The total number of edges in the k chains is $\sum_{i=1}^k (\beta_i - 1) = n - k$, which represents an $(n-k)$ -flow in $G(P_c)$ with the same edges. If there is an element v_i that is not related to any of the other elements in the POSET, v_i will form a trivial chain just by itself. The rest of the $n-1$ elements can form nontrivial chains. The minimum-cost $(n-k)$ -flow will now form $(n-1) - (n-k) = k-1$ nontrivial chains with the minimum total weight. Adding v_i into the solution, we form k disjoint chains with the minimum total weight (v_i contributes no weight to the k -cofamily). Obviously, this result can be extended to k -cofamilies that contain multiple trivial chains. ■

Corollary 2: Let k be the minimum number of registers required to bind all the n variables in the compatibility graph $G_c = (V_c, A_c)$; the solution of the minimum-weighted k -cofamily in the corresponding POSET P_c generates the solution to fulfill the objective of our register binding problem.

Proof: Direct derivation from Theorems 1 and 2. ■

Our task is then to find the minimum-cost $(n-k)$ -flow in the network $G(P_c)$. Its runtime complexity is $O(|E| \log Z(|E| + |V| \log |V|))$ [1], where Z is an upper bound on the largest supply/demand and largest capacity in the network. In our case, $Z = n - k$. After we obtain the minimum-cost flow, each edge with a unit flow in $G(P_c)$, $e = (x_i, y_j)$, represents that variables v_i and v_j should be bound together into the same register. If

there is a flow for $e = (x_i, y_i)$, then it means that v_i occupies a register just by itself. In the example shown in Fig. 7(b), if we have a unit flow on each of the following edges— (x_1, y_2) , (x_2, y_4) , and (x_3, y_5) , the binding solution will be equivalent to the solution shown in Fig. 7(a), i.e., variables v_1, v_2 , and v_4 will be bound to one register, and variables v_3 and v_5 will be bound to another register. In the next section, we will describe how to estimate weights on the edges through cost function formulation.

Cost Function Formulation: In this section, we provide some details for calculating the edge weight w_{ij} if v_i and v_j are to be bound together. A MUX occurs in two situations: 1) It is introduced before a port P of an FU when more than two registers feed data to this port, and 2) it is introduced before a register R when more than two FUs produce results and store them into this register. Different register bindings will produce different multiplexing situations.

Fig. 8 shows an example. Case 1 binds the two variables driven by FUs U_1 and U_2 into two separate registers. By such, it saves a MUX between the connections of U_1/U_2 and their output registers. However, two more MUXes will be required for the connections of the two registers R_1/R_2 to the fan-out FUs (we call them *fan-out_FUs*) U_3 and U_4 . On the other hand, Case 2 binds the two variables from U_1 and U_2 into a single register, and as a result, a MUX is generated between U_1/U_2 and register R . Yet, it is a better solution than Case 1 because there are no MUXes required between R and fan-out_FUs U_3/U_4 . Note that if U_1 and U_2 are actually the same FUs, then there will not be a MUX generated in Case 2, which makes Case 2 an even better solution. However, there are situations where Case 1 is better than Case 2, particularly when U_1 and U_2 are different. A simple case happens when none of the fan-out_FUs in Case 1 requires a MUX on its ports, so it uses one less MUX than Case 2. In the real situation, it is hard to predict which case is better because it all depends on the original DFG data flow, scheduling results, and the FU binding solution. The cost function is defined as follows:

$$w_{ij} = -(N_{\text{mux}} + \alpha \cdot T_{r-f} + \beta \cdot T_{\text{fu}}) \quad (10)$$

where N_{mux} is the number of MUXes saved (or MUXes wasted, i.e., N_{mux} becoming negative) by binding v_i and v_j into a single register (Case 2) rather than not binding them into a single register (Case 1); T_{r-f} is the total number of connections between registers R_1/R_2 and fan-out_FUs; T_{fu} is the total number of fan-out_FUs involved during this attempted binding of v_i and v_j ; and α and β are positive scaling constants (set as 0.25 and 0.15, respectively, through an empirical study). For the example in Fig. 8, $N_{\text{mux}} = 1$, $T_{r-f} = 4$, and $T_{\text{fu}} = 2$. The term T_{r-f} is trying to capture the overall connectivity situation of the fan-out_FUs so that some global optimization criteria can be considered. It is needed because not all of the fan-out_FUs that connect to R_1/R_2 require a MUX on their ports. If its value is large, Case 2 is preferred, i.e., v_i and v_j are to be bound into a single register to reduce the total connectivity of the fan-out_FUs. The term T_{fu} is trying to capture the overall connectivity from another angle, i.e., if more fan-out_FUs are involved, Case 2 is preferred. Nonetheless, N_{mux} is set as the

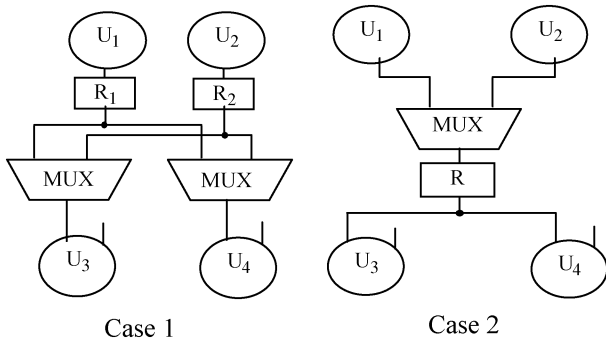


Fig. 8. One example of multiplexing situations.

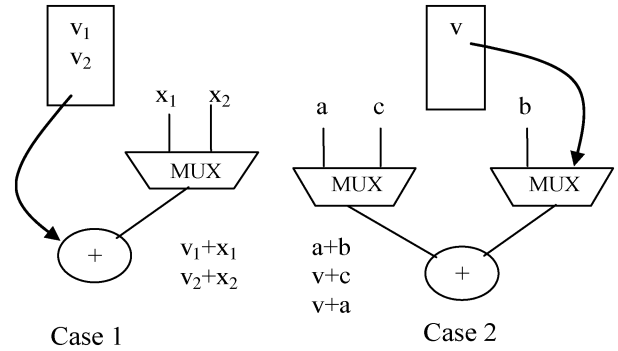


Fig. 10. Operand swapping for the two cases.

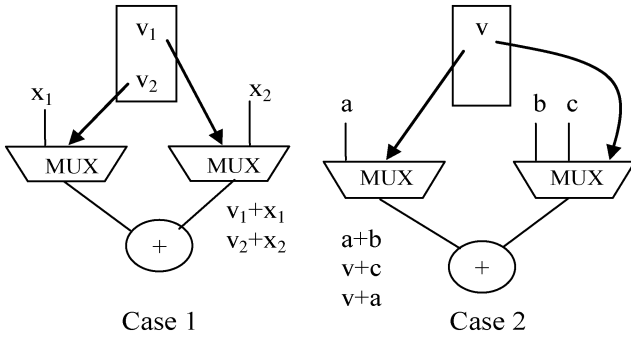


Fig. 9. Cases of a register connecting to both ports on an FU.

overwhelming factor in this cost function because it directly reflects the MUX usage of this binding. The smaller the cost, the better to bind v_i and v_j together. In other words, the larger the N_{mux} , the better.

C. Port Assignment

Port assignment is an important technique for reducing MUX connections between FUs and registers. However, effective heuristics have not been proposed to practically tackle this difficult problem during high-level synthesis. We will apply a greedy algorithm for port assignment, which is shown very effective in our experiments.

In [26], an important lemma proves that finding the minimum connectivity port assignment is equivalent to minimizing the number of input registers that are connected to both ports of an FU U . An optimal solution will be automatically obtained for U if there are no input registers that drive both ports of U . We will use this lemma to guide our port assignment solutions.

We observe two cases where a register is connected to both ports of U . In Fig. 9, the register in Case 1 contains variables v_1 and v_2 , and the operations on FU U are $v_1 + x_1$ and $v_2 + x_2$. Because of the bad port assignments of x_1 and x_2 , this register has to drive both ports of U and renders a total of four connections. The register in Case 2 contains a variable v , and the three operations, together with the current port assignments, force v to drive both ports of U and render five connections.

Fortunately, we observe that the interconnection number of both cases can be reduced using a simple operation of operand swapping. For Case 1, we can swap the port assignments of v_1 and x_1 , while for Case 2, we can swap the port assignments of v and c . The solutions are shown in Fig. 10. In our port assignment

TABLE II
BENCHMARK INFORMATION

Bench marks	No. of PIs	No. of POs	No. of Adds	No. of Mults	Total No. of Edges
chem	20	10	171	176	731
dir	8	8	84	64	314
honda	9	2	45	52	214
mcm	8	8	64	30	252
pr	8	8	26	16	134
steam	5	5	105	115	472
wang	8	8	26	22	134

TABLE III
WIRE LENGTH AND POWER ESTIMATION

Bench marks	After P & R		Estimated		Estimation Error	
	Wire Length	Total Power (W)	Wire Length	Total Power (W)	Wire Length	Total Power
chem	291696	2.15	285553	1.97	-2.1%	-8.6%
dir	141614	1.14	143341	1.08	1.2%	-4.9%
honda	78179	0.63	106212	0.73	35.9%	16.2%
mcm	84140	0.68	70827	0.51	-15.8	-25.3
pr	41080	0.38	37665	0.33	-8.3%	-11.2
steam	174856	1.34	226243	1.54	29.4%	15.1%
wang	37877	0.40	39024	0.32	3.0%	-19.6
Absolute value average:					13.7%	14.4%

algorithm, we first provide a solution done by a random port assignment. Next, we find all the registers that drive both ports of their corresponding FUs and perform operand swapping. There are some situations where operand swapping will not help. For example, in Case 2, if we encounter a series of circular operations such as $a + b$, $b + c$, and $c + a$, then the register has to drive both ports. We check these situations first so that the operand swapping procedure exits early for these situations.

VI. EXPERIMENTAL RESULTS

LOPASS takes in a design and runs through the architectural synthesis stages shown in Fig. 3. The allocation, binding, and scheduling information is back-annotated to the DFG's edges and nodes. The backend of our architectural synthesis system extracts this information to construct the data path and controller. The data path, including instances of FUs, registers, and multiplexers, is generated as a structural VHDL description. The generated FSM controller is in a classical synthesizable RTL

TABLE IV
TOTAL NUMBER OF MUX INPUTS AND RUNTIME OF DIFFERENT ALGORITHMS

Bench marks	Cofamily with pa		Cofamily w/o pa			Bipartite w/o pa			Leftedge w/o pa		
	MUX Ports	Run Time (s)	MUX Ports	Run Time (s)	Cmp %	MUX Ports	Run Time (s)	Cmp %	MUX Ports	Run Time (s)	Cmp %
chem	545	12	554	12	1.7%	703	3	29.0%	735	0	34.9%
dir	204	2	209	2	2.5%	277	0	35.8%	296	0	45.1%
honda	152	0	157	0	3.3%	198	0	30.3%	204	0	34.2%
mcm	146	1	153	0	4.8%	187	0	28.1%	176	0	20.5%
pr	76	0	76	0	0.0%	86	1	13.2%	89	1	17.1%
steam	363	4	370	4	1.9%	483	1	33.1%	500	0	37.7%
wang	84	0	85	0	1.2%	87	0	3.6%	99	0	17.9%
Average					2.2%			24.7%			29.6%

VHDL form. The controller provides control signals for every FU, register, and multiplexer. Register read/write or clock enable/disable operations are also decided by the control signals. We then feed both the data path and control of the design to Synopsys *Design Compiler* for synthesis and mapping. Design Compiler transforms the RTL design into a gate-level netlist in VHDL format. After the VHDL-to-BLIF conversion, the design is fed into *fpgaEva_LP2* [24]. We set *fpgaEva_LP2* to use the same architecture model as that used in LOPASS.

A set of data-intensive benchmarks are used in our experiments. The pure DFG programs are from [30], including several different DCT algorithms, such as *pr*, *wang*, and *dir*, and several DSP programs, such as *mcm*, *honda*, and *chem*. Table II shows the benchmark information. Each node is either an addition/subtraction or a multiplication. We first show the experimental data on the high-level power estimator. We then compare our MUX reduction results to those of a previous algorithm [18]. To evaluate LOPASS, we compare our results with an academic tool SPARK [17]. We also compare LOPASS against commercial high-level synthesis tools Synopsys *Behavioral Compiler* (S-BC) [32] and *Impulse C* [19].

A. Power Estimation

Table III shows the comparison results between our estimated wire length and power and those reported by *fpgaEva_LP2* after placement and routing. We observe that wire length is 13.7% (absolute value average) away from reality. This indicates that Rent's rule-based estimation method is effective to estimate wire length for FPGA designs before layout information is available. Our high-level power estimation also works relatively well with a 14.4% average error (absolute value average). Since behavioral-level estimation is two levels higher than gate-level estimation, we believe that these results are very encouraging.

B. Multiplexer Optimization Results

To examine the effectiveness of our MUX optimization algorithms, we conduct experiments to compare our solutions to those generated by a previously published algorithm in [18]. Table IV shows the total MUX input results for *k-cofamily* with *pa* (port assignment), *k-cofamily* without *pa*, *bipartite* without *pa* [18], and *left edge* without *pa*. The runtime results of these algorithms are also shown. A zero runtime means that it is less than 0.5 s. Columns 6, 9, and 12 show the comparison data using *k-cofamily* with *pa* as the base. We can see that, overall,

TABLE V
ALLOCATION AND SCHEDULING RESULTS OF SPARK/LOPASS

Benchmarks	ADD	MUL	Latency (Cycles)
chem	6/6	10/10	163/57
dir	4/4	8/8	35/31
honda	-/3	-/6	-/29
mcm	4/4	3/3	33/36
pr	2/2	2/2	21/23
steam	-/6	-/10	-/38
wang	2/2	2/2	24/27

TABLE VI
BINDING AND SCHEDULING RESULTS (S-BC/LOPASS)

Bench marks	ADD	MUL	Cycle	Reg No.	Run Time (Min)
chem	28/6	29/10	57/57	163/69	1052/72
dir	9/4	16/8	32/31	75/51	25/25
honda	9/3	14/6	29/29	55/33	40/18
mcm	23/4	6/3	36/36	118/54	21/16
pr	13/2	8/2	24/23	33/31	3/5
steam	20/6	20/10	38/38	113/57	331/29
wang	5/2	8/2	29/27	29/30	5/6

k-cofamily w/o pa, *bipartite w/o pa*, and *left-edge w/o pa* algorithms are 2.2%, 24.7%, and 29.6% worse, respectively, than *k-cofamily with pa*. The gain is mainly coming from register binding because port assignment just provides a small improvement. This shows that our *k-cofamily*-based algorithm is able to reduce the interconnection cost much more effectively than the bipartite-based algorithm.

To evaluate the impact of multiplexer optimization, we carry out two experiments. One takes the RTL outputs of regular LOPASS with MUXes optimized under *Cofamily with pa*. Another takes the RTL outputs of LOPASS under *Leftedge w/o pa*. We then go through *fpgaEva_LP2* evaluation flow for both cases. On average, LOPASS with *Leftedge w/o pa* is 9.2%, 1.5%, and 9.8% worse compared to regular LOPASS in terms of wire length, performance, and power consumption, respectively. Details are omitted due to page limitation.

We also evaluated the effectiveness of our port assignment heuristic. We first computed the upper bound of the total possible reductions of MUX inputs for all the benchmarks by port assignment. This upper bound is obtained assuming that all the

TABLE VII
LUT NUMBER, DELAY, AND POWER COMPARISON BETWEEN S-BC AND LOPASS

Benchmarks	S-BC			LOPASS			Comparison		
	No. of CLBs	Delay (ns)	Power (W)	No. of CLBs	Delay (ns)	Power (W)	No. of CLBs	Delay	Power
chem	11700	92.0	7.478	4267	68.9	2.151	-63.5%	-25.1%	-71.2%
dir	4722	55.2	2.836	2460	51.7	1.140	-47.9%	-6.4%	-59.8%
honda	4156	40.9	1.939	1588	35.9	0.627	-61.8%	-12.3%	-67.7%
mcm	4033	52.7	1.689	1652	41.2	0.679	-59.0%	-21.8%	-59.8%
pr	1935	28.9	0.793	961	35.8	0.377	-50.3%	23.8%	-52.5%
steam	6867	67.6	3.799	2906	52.1	1.340	-57.7%	-22.9%	-64.7%
wang	2292	37.1	0.907	891	33.6	0.403	-61.1%	-9.4%	-55.5%
Average							-57.3%	-10.6%	-61.6%

registers that initially drive both ports of FUs can be changed to only driving a single port through port reassignment. We observe that our heuristic achieved a 54.8% MUX-input reduction of the upper bound value.

C. LOPASS Compared to SPARK

SPARK [17] is a recent behavioral synthesis system focusing on scheduling combined with code transformation techniques. The core scheduling and allocation module contains two parts: the engine combining scheduling with code transformations, and the toolbox containing various compiler passes and transformations.

Although SPARK does not target low-power designs, it is worthwhile to compare the synthesis results with LOPASS for a preliminary evaluation of LOPASS. In this experiment, we intentionally enforce the same resource constraints (numbers of adders and multipliers) for SPARK and LOPASS and check the final scheduling results. We set SPARK to optimize latency, given the resource constraints. The SPARK release used in this experiment is version 1.2 for Linux. In Table V, the data before (after) the forward slash “/” is generated by SPARK (LOPASS). For *honda* and *steam*, SPARK fails to produce results. For other cases, SPARK sometimes produces much worse latency (e.g., 163 versus 57 for *chem*) or slightly better results (e.g., 33 versus 36 for *mcm*). Overall, LOPASS is 9.1% better in terms of latency optimization.

D. LOPASS Compared to Behavioral Compiler

To further validate LOPASS, we compare LOPASS with an early commercial tool—Synopsys Behavioral Compiler (S-BC). We set the *high optimization effort* option for S-BC so that it returns the best solutions. S-BC does not consider the specific features of FPGAs. Table VI shows the allocation and scheduling results from both S-BC and LOPASS. The data before (after) the forward slash “/” is generated by S-BC (LOPASS). We can observe that LOPASS is able to schedule each design using the same or smaller number of clock cycles.

LOPASS runs much faster for the two large designs: *chem* and *steam* (experiments carried out on a 750-MHz SunBlade 1000 machine). LOPASS also uses a much smaller number of resources (adders and multipliers). There is no easy way to collect the multiplexer usage for S-BC’s solutions, so it is not listed in these tables. At this point, we cannot directly claim that LOPASS is much better on area and power because LOPASS

TABLE VIII
IMPULSE C SYNTHESIS RESULTS

Bench mark	Cyc.	Resource Usage for Datapath					
		16b ADD	32b ADD	16b MUL	32b MUL	16b Reg	32b Reg
dir	16	84	0	64	0	82	0
honda	14	45	0	52	0	64	0
mcm	8	64	0	30	0	74	0
pr	7	26	0	16	0	76	0
wang	12	18	8	14	8	48	4

TABLE IX
LOPASS SYNTHESIS RESULTS

Bench mark	Cyc.	Resource Usage for Datapath			
		24b ADD	18b MUL	24b Reg	MUX Inputs
dir	16	8	7	49	207
honda	14	5	7	36	149
mcm	8	13	10	54	171
pr	7	6	6	33	89
wang	12	3	4	30	85

only uses FUs of fixed bit widths. However, S-BC usually uses FUs of different bit widths for constant handling and timing optimization. Therefore, although S-BC uses more multipliers and adders than LOPASS, the sizes of these operators can be smaller than those used in LOPASS. We need to verify the final area, power, and delay results using our evaluation flow.

Table VII shows the area, delay, and power comparison results. The RTL designs generated from both S-BC and LOPASS for the same benchmarks are evaluated by *fpgaEva_LP2*. Area is the number of CLBs used in the design. On average, our solution reduces the number of CLBs by 57.3% and reduces the total power consumption by 61.6% compared to S-BC’s solution. We also reduce the critical-path delay by 10.6%.

E. LOPASS Compared to Impulse C

We also compare our results with that of a popular C-to-FPGA commercial tool, i.e., *Impulse C* [19]. To use *Impulse C*, an application needs to be partitioned into software (SW) and hardware (HW) parts in C language. The SW part is responsible for preparing inputs for HW and gathering outputs from HW. The HW part implements the compute-intensive portion. The SW and HW parts communicate through a dedicated bus

TABLE X
QUARTUS II FPGA IMPLEMENTATION RESULTS FOR IMPULSE C AND LOPASS

Benchmarks	Impulse C							LOPASS						LOPASS vs. Impulse C					
	Clock period (ns)	Total LEs	Dedicated registers	Multiplier (9-bit)	Dynamic power (mw)	Static power (mw)	Total power (mw)	Clock period (ns)	Total LEs	Dedicated registers	Multiplier (9-bit)	Dynamic power (mw)	Static power (mw)	Total power (mw)	Dynamic power	Total power	Delay	9-bit multiplier	LE
dir	6.81	2819	1104	120	425.0	92.4	517.4	7.52	1981	209	14	176.8	92.6	269.5	-58.4%	-47.9%	10.4%	-88.3%	-29.7%
honda	5.70	1825	644	100	287.7	92.1	379.8	7.07	1233	111	14	127.1	91.9	219.0	-55.8%	-42.3%	24.1%	-86.0%	-32.4%
mcm	6.65	2343	1258	60	255.0	92.0	347.1	6.47	2192	321	20	212.9	92.7	305.6	-16.5%	-11.9%	-2.7%	-66.7%	-6.4%
pr	5.41	1478	1196	32	152.3	91.8	244.1	6.61	1012	152	12	79.4	92.2	171.5	-47.9%	-29.7%	22.1%	-62.5%	-31.5%
wang	6.17	1568	1064	44	118.4	91.7	210.1	6.49	947	85	8	68.9	92.1	161.1	-41.8%	-23.4%	5.2%	-81.8%	-39.6%
Ave.															-44.1%	-31.1%	11.8%	-77.1%	-27.9%

(up to 32 b in width). Impulse C can automatically generate first-in–first-out or memory blocks to temporarily hold input data when necessary.

Table VIII shows the results from Impulse C. It can be seen from the table that much more adders and multipliers are needed in these solutions from Impulse C. It is obvious that Impulse C's objective is not resource sharing or register sharing but running the operations in a streaming fashion. As a result, each definition of variable or operation is turned into a dedicated register or FU. The total numbers of adders and multipliers in the solution are equal to the numbers of additions and multiplications in the original C specification, respectively. Impulse C does have bit-width analysis capability, so some adders/multipliers may use smaller bit widths. Note that we did not try the largest benchmarks (*steam* and *chem*) because they simply follow the same trend, and the nonsharing solutions of these largest benchmarks with hundreds of adders and multipliers are already on the boundary of exceeding the logic capacity of many FPGA chips. Table IX shows the results of LOPASS, which uses the same clock cycles as those in Table VIII to have a fair comparison.

The synthesis results from Impulse C and LOPASS are evaluated using Quartus II, a commercial FPGA design environment from Altera. A 536-I/O Cyclone III FPGA device, namely, EP3C40F780C6, is chosen as the target device because it provides abundant LEs, I/Os, and 252 hardwired 9-b multipliers. Aggressive timing requirements are set to exert high-effort design in Quartus II. For each benchmark, we took the RTL outputs, including both data path and controller, of Impulse C and LOPASS and ran through synthesis, clustering, placement, and routing to fit into the FPGA chip. Then, we used the power estimator available in Quartus II for power estimation. Table X is the summary of the comparison results. Impulse C's solution does not generate MUXes, so it offers a better critical path delay compared to LOPASS' results. However, with an 11.8% penalty on clock period, LOPASS can accomplish the same task with far less resources. On average, LOPASS saves 9-b multipliers by 77.1% and LEs by 27.9%. The savings on power are also significant: 44.1% and 31.1% for dynamic and total powers, respectively. Note that, for Impulse C, only the HW portion is used to do the comparison.

Overall, in Section VI, we have demonstrated that LOPASS provides a significant amount of improvement for FPGA

designs during high-level synthesis to optimize area, power, and latency. We believe that the following features lead to the promising results of LOPASS: 1) accurate FPGA architecture and CAD design flow modeling; 2) well-designed optimization engine with FPGA interconnect and steering logic power estimation; and 3) effective interconnect optimization through multiplexer reduction.

VII. CONCLUSION AND FUTURE WORK

We have presented a low-power architectural synthesis system, i.e., LOPASS, for FPGA designs with interconnect power estimation and optimization. It includes three major components: 1) a flexible FPGA high-level power estimator considering interconnect power; 2) a simulated-annealing-based optimization engine; and 3) a postprocessing *k-cofamily*-based register binding algorithm and an efficient port assignment algorithm for multiplexer optimization. Overall, LOPASS is 61.6% better on power consumption and 10.6% better on clock period compared to that of an early commercial high-level synthesis tool Synopsys Behavioral Compiler. Compared to a current commercial C-to-FPGA tool, i.e., Impulse C, LOPASS is 31.1% better on power consumption with an 11.8% penalty on clock period. In the future, detailed high-level glitch power modeling will be explored. FPGA architecture evaluation using LOPASS will also be explored.

REFERENCES

- [1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*. Englewood Cliffs, NJ: Prentice-Hall, 1993, sec. 10.2.
- [2] "Stratix Device Handbook" Altera Corporation, San Jose, CA [Online]. Available: http://www.altera.com/literature/hb/stx/stratix_handbook.pdf
- [3] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA: Kluwer, 1999.
- [4] A. Bogliolo, L. Benini, B. Riccò, and G. De Micheli, "Efficient switching activity computation during high-level synthesis of control-dominated designs," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug. 1999, pp. 127–132.
- [5] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," in *Proc. Asia South Pacific Des. Autom. Conf.*, Jan. 2004, pp. 68–73.
- [6] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug. 2003, pp. 134–139.

- [7] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for FPGAs," in *Proc. Asia South Pacific Des. Autom. Conf.*, Jan. 2007, pp. 529–534.
- [8] J. Cong and C. L. Liu, "On the k -layer planar subset and topological via minimization problems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 8, pp. 972–981, Aug. 1991.
- [9] J. A. Davis, V. K. De, and J. Meindl, "A stochastic wire-length distribution for gigascale integration (GSI)—Part I: Derivation and validation," *IEEE Trans. Electron Devices*, vol. 45, no. 3, pp. 580–589, Mar. 1998.
- [10] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
- [11] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in datapath synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 8, no. 7, pp. 768–781, Jul. 1989.
- [12] R. P. Dilworth, "A decomposition theorem for partially ordered set," *Ann. Math.*, vol. 51, pp. 161–166, 1950.
- [13] W. E. Donath, "Placement and average interconnection lengths of computer logic," *IEEE Trans. Circuits Syst.*, vol. CAS-26, no. 4, pp. 272–277, Apr. 1979.
- [14] A. A. Duncan, D. C. Hendry, and P. Gray, "An overview of the COBRA-ABS high level synthesis system for multi-FPGA systems," in *Proc. Symp. FPGAs Custom Comput. Mach.*, 1998, pp. 106–115.
- [15] M. Feuer, "Connectivity of random logic," *IEEE Trans. Comput.*, vol. C-31, no. 1, pp. 29–33, Jan. 1982.
- [16] C. Greene and D. Kleitman, "The structure of Sperner k -family," *J. Comb. Theory, Ser. A*, vol. 20, pp. 41–68, 1976.
- [17] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Norwell, MA: Kluwer, 2004.
- [18] C. Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *Des. Autom. Conf.*, 1990, pp. 499–504.
- [19] Impulse Accelerated Technologies, Kirkland, WA, "Impulse C," [Online]. Available: <http://www.impulsec.com/>
- [20] P. Kollig and B. M. Al-Hashimi, "Simultaneous scheduling, allocation and binding in high level synthesis," *Electron. Lett.*, vol. 33, no. 18, pp. 1516–1518, Aug. 1997.
- [21] E. Kusse and J. Rabaey, "Low-energy embedded FPGA structures," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug. 1998, pp. 155–160.
- [22] B. Landman and R. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. Comput.*, vol. C-20, no. 12, pp. 1469–1479, Dec. 1971.
- [23] F. Li, D. Chen, L. He, and J. Cong, "Architecture evaluation for power-efficient FPGAs," in *Proc. ACM Int. Symp. FPGA*, Feb. 2003, pp. 175–184.
- [24] F. Li, Y. Lin, L. He, D. Chen, and J. Cong, "Power modeling and characteristics of field programmable gate arrays," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 11, pp. 1712–1724, Nov. 2005.
- [25] C. L. Liu, *Elements of Discrete Mathematics*. New York: McGraw-Hill, 1977.
- [26] B. Pangrle, "On the complexity of connectivity binding," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 11, pp. 1460–1465, Nov. 1991.
- [27] M. Pedram, "Low power design methodologies and techniques: An overview," Mar. 1999. [Online]. Available: <http://atrk.usc.edu/~mas-soud>
- [28] L. Shang, A. Kaviani, and K. Bathala, "Dynamic power consumption in Virtex-II FPGA family," in *Proc. Int. Symp. Field-Program. Gate Arrays*, Feb. 2002, pp. 157–164.
- [29] A. Singh and M. Marek-Sadowska, "Efficient circuit clustering for area and power reduction in FPGAs," in *Proc. ACM Int. Symp. FPGA*, Feb. 2002, pp. 59–66.
- [30] M. B. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, no. 1, pp. 2–19, Mar. 1995.
- [31] D. Stroobandt and J. V. Campenhout, "Accurate interconnection length estimations for predictions early in the design cycle," *VLSI Des.—Special Issue Phys. Des. Deep Submicron*, vol. 10, no. 1, pp. 1–20, 1999.
- [32] Synopsys, San Jose, CA, "Synopsys," [Online]. Available: http://www.synopsys.com/products/products_matrix.html
- [33] M. Vasilko and D. Ait-Boudaoud, "Scheduling for dynamically reconfigurable FPGAs," in *Proc. Int. Workshop Logic Architecture Synthesis*, 1995, pp. 328–336.
- [34] F. G. Wolff, M. J. Knieser, D. J. Weyer, and C. A. Papachristou, "High-level low power FPGA design methodology," in *Proc. IEEE Nat. Aerosp. Conf.*, 2000, pp. 554–559.
- [35] M. Xu and F. J. Kurdahi, "Layout-driven high level synthesis for FPGA based architectures," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, 1998, pp. 446–450.

Deming Chen (S'01–M'05) received the B.S. degree in computer science from University of Pittsburgh, PA, in 1995, and the M.S. and Ph.D. degrees in computer science from University of California at Los Angeles, in 2001 and 2005, respectively.

He is a technical committee member for a series of conferences and symposia. His current research interests include nano-systems design and nano-centric CAD techniques, FPGA synthesis and physical design, high-level synthesis, microprocessor architecture design under process/parameter variation, and reconfigurable computing. He is a TPC subcommittee chair for ASPDAC'09-10 and a CAD Track co-chair for ISVLSI'09.

Dr. Chen was a recipient of the Achievement Award for Excellent Teamwork from Aplus Design Technologies in 2001, the Arnold O. Beckman Research Award from UIUC in 2007, the NSF CAREER Award in 2008, and the ASPDAC Best Paper Award in 2009. He is included in the List of Teachers Ranked as Excellent in 2008. He is an Associated Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS.

Jason Cong (S'88–M'90–SM'96–F'00) received the B.S. degree in computer science from Peking University, Peking, China, in 1985, the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign, in 1987 and 1990, respectively.

Currently, he is a Chancellor's Professor at the Computer Science Department of University of California, Los Angeles, and a co-director of the VLSI CAD Laboratory. He also served as the department chair from 2005 to 2008. His research interests include computer-aided design of VLSI circuits and systems, design and synthesis of system-on-a-chip, programmable systems, novel computer architectures, nano-systems, and highly scalable algorithms.

Dr. Cong was a recipient of a number of awards and recognitions, including the Ross J. Martin Award for Excellence in Research from the University of Illinois at Urbana-Champaign in 1989, the NSF Young Investigator Award in 1993, the Northrop Outstanding Junior Faculty Research Award from UCLA in 1993, the ACM/SIGDA Meritorious Service Award in 1998, and the SRC Technical Excellence Award in 2000. He also received four Best Paper Awards. He was elected to an IEEE Fellow in 2000 and ACM Fellow in 2008.

Yiping Fan (S'02–M'06) received the B.S. degree in electrical engineering and the M.S. degree in computer science from Tsinghua University, Beijing, China, in 1998 and 2001, respectively, and the Ph.D. degree in computer science from University of California, Los Angeles, in 2006.

Currently, he is a cofounder and director of engineering in synthesis infrastructure of AutoESL Design Technologies, Inc. He has published over 20 publications in the research area of electronic design automation.

Lu Wan (S'08) received the B.S. degree in electrical engineering and the M.S. degree in computer science and engineering from Jiaotong University, Shanghai, China, in 2001 and 2004, respectively. He is currently pursuing the Ph.D. degree in the Electrical and Computer Engineering Department, University of Illinois, Urbana-Champaign.

He was with IBM China Research Lab as a R&D Engineer from 2004 to 2006. His research interests include VLSI design for application acceleration, reconfigurable computing, and CAD techniques for logic synthesis and physical design.