High-Level Power Estimation and Low-Power Design Space Exploration for FPGAs

Deming Chen
Department of ECE
University of Illinois, Urbana-Champaign
dchen@uiuc.edu

Jason Cong, Yiping Fan, Zhiru Zhang
Computer Science Department
University of California, Los Angeles
{cong, fanyp, zhiruz}@cs.ucla.edu

ABSTRACT

In this paper, we present a simultaneous resource allocation and binding algorithm for FPGA power minimization. To fully validate our methodology and result, our work targets a real FPGA architecture — Altera Stratix FPGA [2], which includes generic logic elements, DSP cores, and memories, etc. We design a high-level power estimator for this architecture and evaluate its estimation accuracy against a commercial gate-level power estimator — Quartus II PowerPlay Analyzer [1]. During the synthesis stage, we pay special attention to interconnections and multiplexers. We concentrate on resource allocation and binding tasks because they are the key steps to determine the interconnections. We use a novel approach to explore the design space. Experimental results show that our high-level power estimator is 8.7% away from PowerPlay Analyzer. Meanwhile, we are able to achieve a significant amount of power reduction (32%) with better circuit speed (16%) compared to a traditional resource allocation and binding algorithm.

1. INTRODUCTION

The basic problem of high-level synthesis is the mapping of a behavioral description of a digital system into an RTL design consisting of a datapath and a control unit. A datapath is composed of three types of components: functional units (e.g., ALUs, multipliers, and shifters), storage units (e.g., registers and memory), and interconnection units (e.g., buses and multiplexers). The control unit is specified as a finite state machine, which controls the set of operations for the datapath to perform during every control step. The high-level synthesis process mainly 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, registers, or interconnection units) are needed; binding binds operations, variables, or data transfers to these resources.

Traditionally, people are more concerned with area and power of functional units and registers. As technology advances, the area and power of multiplexers and interconnects have by far outweighed the area and power of functional units and registers, especially for FPGA architectures. Studies show that interconnects contribute 70-80% of the total area [27] and 75-85% of the total power [19][20] in FPGAs. Multiplexers are particularly expensive for FPGA architectures. It is shown that the delay and power data of a 32-to-1 multiplexer are almost equivalent to a 18-bit multiplier in 0.1um technology in FPGA designs [6][7]. In general, smaller number of functional units or registers allocated but with larger number of wide multiplexers and larger amount of interconnects may lead to a completely unfavorable solution for both performance and power. To tackle this increasingly alarming problem, it will require an efficient search engine to explore a sufficiently large solution space considering multiple constraining factors, such as resource allocation and binding, MUX generation, and interconnection generation, for optimizing performance or power, or study the tradeoff between them.

Although low-power high-level synthesis for ASICs is an old topic, high-level synthesis for FPGA power minimization has not been widely studied. We are only aware of one previous work [7], where the optimization goal is to minimize power of FPGA designs under performance/latency constraints. The authors adopted a simulated annealing-based algorithm, which carried out high-level synthesis subtasks simultaneously. However, the delay model in [7] did not consider multiplexer delay, which could represent a significant portion of the critical path delay for FPGA chips. Also, [7] only worked on data-flow graphs (DFGs), and it did not model existing commercial FPGAs.

In this work we present a novel design space exploration engine, *xPlore-Power*, for FPGA power minimization. We concentrate on resource allocation and binding tasks because they are the key steps to determine the interconnections during high-level synthesis. To fully validate our methodology and result, we target a real FPGA architecture — Altera Stratix architecture [2], which includes generic logic elements, DSP cores, and different types of memories, etc. We design a high-level power estimator for this architecture and verify that its power estimation result is very close to that reported by Altera's gate-level power estimator — Quartus II PowerPlay Analyzer [1]. We form, propagate and prune binding/allocation solution points guided by our power and delay estimation. During this process, we pay attention to interconnects and multiplexers to control their power consumption and delay. Eventually, we generate a design solution curve, which can provide ideal solution points with low power and high performance.

The rest of this paper is organized as follows. In Section 2 we present related work. Section 3 provides definitions and problem formulation. Section 4 presents CDFG simulation, and power and delay estimation. Section 5 presents detailed description of our xPlore-Power algorithm. Section 6 presents experimental results, and Section 7 concludes this paper.

2. RELATED WORK

There is extensive literature on binding and allocation problems for high-level synthesis [10][11]. The previous work can be roughly categorized into two major groups. The first group solves register binding and functional unit binding separately. Representative algorithms include clique partitioning [28], weighted bipartite-matching [15], network flow [5][12], and *k*-cofamily [6]. The challenge for these approaches is how to achieve global optimization. The second group tries to address the global optimality and performs simultaneous functional unit and register binding. Representative algorithms include simulated annealing [7][8][18], simulated evolution [21], and ILP (integer linear programming) [13][26]. Since the subtasks of high-level synthesis are highly interrelated, simultaneous optimization approaches try to consider all the involved optimization parameters together and explore the combined solution space for overall better results. One concern for these algorithms is their scalability towards optimizing large designs. There is some work that carries out simultaneous optimization one control step at a time [16][23]. Although this approach may have better runtime, it could lose the global optimization opportunity because it has to commit to a binding solution for each control step, which only represents the local optima.

Most of the work mentioned above is for data-dominated behaviors, normally found in digital signal processing and image processing applications. For control-flow intensive behaviors, frequently found in network-centric systems, different optimization techniques are required to handle branch and loop conditions. There are mainly two approaches to address the hierarchical structure in the design. One approach is to process each basic block separately (there are no conditions and loops in a basic block, thus an easier problem to solve), and then handle the control flow between these blocks to reduce cost or improve performance [22][30]. The other approach is to optimize the whole design directly on top of an internal representation (either hierarchical or flattened) for both datapath and control flow [14][25]. Our work belongs to the latter category.

3. DEFINITIONS AND PROBLEM FORMULATION

3.1 Definitions

State transition diagrams, or STGs, are general models to describe the behavior of sequential circuits. We use STGs to hold the scheduling results from control data flow graph (CDFG), which is the input of our high-level synthesis system. To describe our version of STG representation and its execution semantics, we have the following definitions.

A *dataflow* represents the data produced by some *operation* and consumed by others. An *operation* is an atomic computation of the design behavior. For example, an *addition* operation consumes two dataflows *a* and *b*, and produces their sum to dataflow *c*. A *condition* is a Boolean expression of Boolean dataflows. It can be dynamically evaluated as true or false during the STG execution. The *lifetime* of each dataflow (or operation) in the STG is the time during which the dataflow (or operation) is active and is defined by an interval [birth time; death time]. Two dataflows (or operations) are *compatible* with each other if their lifetimes do not overlap. For example, if the birth time of a dataflow a_1 is earlier than the death time of a_2, a_1 and a_2 are not compatible. Two compatible dataflows (operations) can share the same register (functional unit).

The STG also contains a set of states *S* and a set of transitions *X*. A state contains a set of *operations*, which should be accomplished within one clock cycle in the resulting circuit. A transition connects a source state to a sink state, associated with a *condition.* STG provides a general representation of scheduling results from various behavioral domains. It is a natural representation of control flow-based scheduling results.

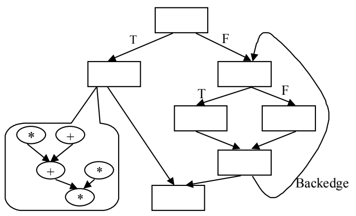

Figure 1: Two-level control data flow graph

We use a two-level CDFG representation for our input design. The first-level CDFG is a control flow graph (CFG). Each node corresponds to a basic block. The edges represent the control dependencies between the basic blocks. Each basic block contains one operation producing the control signal. If there are more than two successors, i.e., *if-then-else* or *switch* statements, the labels on the control edges indicate the values for the respective branches to be taken. A *back* control edge indicates that there is a loop between the source basic block and the destination basic block. The source basic block and the destination basic block of a back edge can be the same, which indicates that the loop only crosses one basic block. At the second level, each basic block has a pure data flow graph (DFG) representation, which contains a set of operation nodes and edges (dataflows) that represent data dependencies among operation nodes. Figure 1 shows one example. After scheduling, each CDFG has a corresponding STG to hold its scheduling result.

3.2 Problem Formulation

High-level synthesis started from a STG essentially is a resource allocation and binding (or sharing) problem, i.e., determine the numbers of functional units and registers, and share functional units among compatible operations and registers among compatible dataflows. These optimization steps have dramatic impacts on the final design quality. Careless allocation and binding will result in unaffordable interconnection resource and multiplexer usage (multiplexers are used to route data and control signals in the design), dropping down the final circuit frequency and increasing the total power. Unfortunately, binding for optimizing interconnection is known to be an NP-hard problem [24]. Even binding in a general STG to minimize resource counts is a difficult problem. Meanwhile, minimizing a single objective number, e.g., interconnection unit, functional unit, or register count, can not guarantee high design quality because these design metrics are interrelated. Therefore, instead of using resource count as the objective function, we use realistic measurements, namely performance and power, to guide our optimization. Performance is usually measured as the latency of the execution, i.e., the product of the execution path-length and the cycle time. Since the path-length is totally determined by the scheduling and thus fixed in the STG, we need only care about the frequency the final design can achieve. Therefore, our synthesis problem can be formulated as follows:

Given: A CDFG *G* and its STG *G'*

Tasks: construct a datapath architecture, in which every functional unit is bound to a set of operations, and every register is bound to a set of dataflows.

Objectives: maintain behavior correctness and optimize power and performance for the design on a target FPGA.

4. POWER AND DELAY MODELING

To efficiently search the solution space during resource allocation and binding, we need a fast and accurate high-level power and performance estimator to guide the process. We first present an efficient switching activity calculator using CDFG simulation. We then present our power characterization method for one type of commercial FPGAs — Altera Stratix FPGAs [2]. We would like to emphasize that similar method can be applied to other types of FPGAs from other FPGA vendors as well. Finally, we present our resource characterization method to estimate the area and speed of different functional units and multiplexers for Stratix.

4.1 CDFG Simulation and Switching Activity Estimation

We carry out test vector-based CDFG functional simulation. The simulation process is iterative. For each iteration, a set of test vectors arrive on the primary inputs of the CDFG. These values will follow the control and data flows in the graph and propagate through the graph until they reach the outputs of the CDFG. Then, another set of vectors arrive for the next iteration. During the propagation, the data get operated on the operators within the basic blocks and then passed on to branches or loops determined by conditions. Data can also be loaded from or stored into the memories. During this simulation, we can profile the CDFG and collect useful information for calculating switching activities, block visiting probabilities, worst-case latency, etc.

For switching activity calculation, we extend a method published in [4], which performs simulation just once at the beginning and computes switching activities for any legal binding without repeating simulations afterwards. We add loop support in the algorithm. The handling for operations not in loops is the same as the method in [4].

Let ($PI^1 \rightarrow PI^2 \dots \rightarrow PI^K$) be a sequence of stimuli enforced on the primary inputs of the CDFG G. By performing functional simulation on G, with primary input stimulus PI^j ($1 \le j \le K$), we can obtain input bit vector I_i^j for operation O_i ($1 \le i \le N$). 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 functional unit U, let ($O_1 \rightarrow O_2 \dots \rightarrow O_N$) be the bound operations in the execution order. Suppose all of these operations are in the same loop with a loop iteration upper bound B.[1] We define $I_i^{j(x)}$ to represent the input bit vector for operation O_i when the simulation takes primary input stimulus PI^j and reaches loop iteration x ($1 \le x \le B$) for O_i. The toggle count between $C_{in}(O_i, O_{i+1})$ and $C_{in}(O_N, O_1)$ under this primary-input stimulus sequence, is then defined as follows:

$$C_{in}(O_i, O_{i+1}) = \sum_{j=1}^{K} \sum_{x=1}^{B} D_H(I_i^{j(x)}, I_{i+1}^{j(x)})$$

$$C_{in}(O_N, O_1) = \sum_{j=1}^{K} \sum_{x=1}^{B-1} D_H(I_N^{j(x)}, I_1^{j(x+1)}) + \sum_{j=1}^{K-1} D_H(I_N^{j(B)}, I_1^{(j+1)(1)})$$

where $1 \le i < N$, and $D_H(X, Y)$ represents the Hamming Distance between bit vectors X and Y. Notice that $C_{in}(O_N, O_1)$ represents the toggle count between O_N and O_1 when the execution finishes O_N and begins O_1 again. It contains two terms. The first one represents the switches from O_N to O_1 within the same loop (but different loop iteration). The second one represents the switches from the end of the previous loop (when it reached the loop upper bound) to the beginning of the new round for the loop (it starts from loop iteration 1 again) when the simulation takes a new primary input stimulus PI^{j+1}. *Transition probability* (or switching activity) P_{in} of the inputs of U is the ratio of the number of bit flips observed on its inputs between cycles over the maximum possible number of bit flips. It is formally defined as

$$P_{in} = \frac{\sum_{i=1}^{N-1} C_{in}(O_i, O_{i+1}) + C_{in}(O_N, O_1)}{2 \times Bit\_width \times (N \times K \times B - 1)}$$

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 for looking up when calculating the P_{in} for every possible binding solution afterwards. We skipped the details here. Once we have switching activities on U, we can use them to derive the switching activities for multiplexers and registers connected to U.

4.2 Power Estimation for Stratix FPGA

Since we target Stratix FPGA device families, we need to deal with Stratix-specific features in our high-level power estimation. Altera provides a spread-sheet-based Early Power Estimator [3], where users can specify switching activities, Fmax, usages of various components (e.g., logic elements, memories, DSPs, I/O ports), and other related information to estimate the total power of Altera's FPGAs in early design stages. We take advantage of this estimator and use it as a resource characterization tool to examine the power consumption of various components in a Stratix FPGA. Table 1 shows some details of our characterization. We use Fmax = 100 Mhz and toggle rate (switching activity) = 100% for the purpose of power characterization. The actual power values for the components will be adjusted based on the estimated Fmax and switching activity during synthesis. Some formulae are listed below:

$$P_{resource} = S_{resource} \cdot A_{resource} \cdot P_{LE} \cdot (Fmax / 100)$$

$$P_{DSP} = 1.23 \cdot S_{DSP} \cdot Bitwidth \cdot (Fmax / 100)$$

$$P_{IO} = 19.31 \cdot S_{IO} \cdot (Fmax / 100)$$

where *resource* represents those components that can be implemented using logic elements (LEs) on the FPGA. These resources can include adders/subtractors, multiplexers, shifters, etc. $S_{resource}$, S_{DSP}, and S_{IO} are the estimated switching activities for the resource, DSP, and I/O respectively; $A_{resource}$ is the estimated number of LEs for the resource when it is implemented in the Stratix FPGA (to be covered in the next subsection); and P_{LE} is either 0.04mW for adders/subtractors, or 0.12mW for random logic or other generic logic. In general, multiplications are realized in the DSP blocks available on the FPGA chip. The power consumed in a DSP core is proportional to the number of DSP outputs (or the bitwidth of the multiplier implemented by the DSP core). P_{IO} is the power of a single I/O pin. We can also calculate clock power, which is related to the number of flip-flops or DSP cores the clock drives.

Elements	# of LEs	Est'ed P (mW)
Logic Element (LE)	1	0.12
LE with carry	1	0.04
DSP	per output	1.23
I/O	1	19.31

Table 1: Power consumption of various FPGA components (Fmax = 100; Toggle rate = 100%)

4.3 Resource Characterization for Stratix FPGA

In general, given the target FPGA architecture, the final area and delay of a functional unit and/or a multiplexer are largely determined by the total number of input operands and the precision (i.e., bitwidth) of the calculation. In this work we take a curve-fitting approach to model the timing/area characteristics of the functional units and multiplexers. We vary the precision for each functional unit (written in RTL code), and we also vary the number of inputs for the multiplexers. We run through the Altera Quartus II RTL synthesis and physical design tool to obtain a set of frequency and resource usage results on Stratix device. After all the data points are collected, we use the curve fitting tool in MATLAB to derive the best-fit area and delay curves.

The area estimation functions for the multiplexers and several commonly occurring arithmetic units are listed in Table 2. Note that due to the high regularity of the FPGA device, the final resource usages of most operations are very predictable and their estimation functions can be expressed in close-form equations. For example, the resource usage (i.e., LEs) of an adder/subtractor is equivalent to its bitwidth. The reason is that Stratix FPGAs embed dedicated carry-select chains in the fabric so that a fast N-bit carry-select adder/subtractor can be efficiently implemented using exactly N logic elements (each contains a 4-input lookup table). Table 2 also shows that the multiplication may require several on-chip hard-core DSP multipliers. In this case, the area curve is discrete as the finest precision of a DSP block is 9×9. To acquire the delay estimation curve, we also perform curve fitting for various operations under different precisions. The results are shown in Table 3. Note that for multiplexers, since we cannot get a close-form equation with both N (precision) and K (input operand count), we only list the formula for the 8-to-1 configuration here. We omit the complete data for other configurations due to space limit. We can observe that the area and delay of multiplexers are significant. For example, when N = 24, an 8-to-1 multiplexer will use up 120 LEs and contribute 3ns on delay, where an adder/subtractor only occupies 24 LEs and contributes

[1] We handle cases when these operations are not in the same loop as well. Details are not shown due to space limit.

2.4ns on delay. This motivates an interconnection-centric design method as proposed in this paper.

Operation	Resource	Usage
Add/Subtract	LE	*N*
Bitwise and/or/xor	LE	*N*
Compare (□□□)	LE	*round*0.67**N*+0.62
Shift (with variable shift distance)	LE	*round*(0.045*N^2+3.76**N*–8.22)
Multiply	DSP9x9	*N* □ 18: □*N*/9□ *N* □ 36: □*N*/18□
Multiplexer	LE	*N***round*(0.67**K*)

Table 2: Area estimation functions for common operations on Altera Stratix FPGAs (*N*: bitwidth; *K*: number of input operands)

Operation	Delay (*ns*)
Add/Subtract	0.024**N*+1.83
Bitwise and/or/xor	< 2
Compare (□□□)	0.014**N*+2.14
Shift (with variable shift distance)	4.3*10^{-5}*N^3–5*10^{-3}*N^2+0.24**N*+0.93
Multiply	*N* □ 9: □□ *N* □ 18: □□□ *N* □ 36: 7.69
Multiplexer (8-to-1)	9.8*10^{-5}*N^3–7.4*10^{-3}*N^2+0.2**N*+1.07

Table 3: Delay estimation functions for common operations on Altera Stratix FPGAs

5. SIMULTANEOUS ALLOCATION AND BINDING

Due to the difficulty of resource allocation and binding with interconnection consideration, it naturally calls for efficient solution space searching and pruning techniques. To enable such searching and pruning, we use a *solution point* to reflect a unique allocation/binding implementation. Each solution point is represented by a pair *[power, delay]*, which contains the estimated power and delay for the datapath implemented through this solution point. We form, propagate and prune these solution points to search for ideal allocation/binding solutions.

Because functional unit and register allocation and binding are interrelated, we adopt an iterative approach. We first carry out a trivial register binding, where each dataflow occupies its own register. This solution does not generate multiplexers in front of registers. After this, we carry out the allocation and binding for functional units and registers separately and iteratively so one allocation and binding is built upon another until the final quality of result converges. In general, we just carry out one such iteration because we observe a single iteration is already generating a good result that is very close to the final one.

We will use functional unit allocation and binding as an example and the same principle is applied for registers. Figure 2 shows a simple example. Figure 2(a) is a STG, where nodes represent states and edges represent transitions. *C1* and *C2* are conditions. The numbers in the states represent operations. Suppose 1, 2, 3, 4, and 5 are multiplications, and 6 is addition. Multiplications will be implemented by multipliers, and addition and comparison operations can be implemented by ALUs. Figure 2(b) is called a *global compatibility graph*, where each node is an operation, and the edge between two nodes represents that those two operations are compatible with each other. Notice Figure 2(b) actually contains two sub-graphs: one for multiplication and one for ALU. Nodes 2 and 3 are not compatible with each other because they are in the same state in Figure 2(a). After we have the global compatibility graph we can start our functional unit allocation and binding procedure. The procedure will visit each operation in the global compatibility graph and carry out solution space exploration along the way. For illustration purpose, we assume that there are just four operations 1, 2, 3, and 4 in the design, and we process the nodes according to that order. Note that we already have a register allocation and binding solution before this. Therefore, we can build a real datapath when we examine each solution point during the exploration. We present some details next.

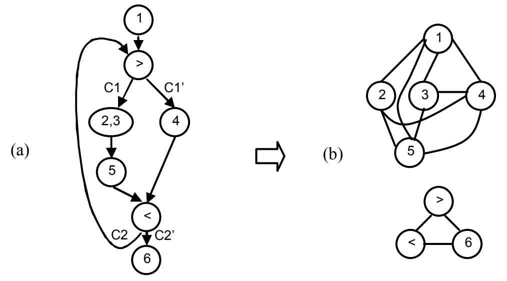

Figure 2: A STG and its global compatibility graph

When we reach node 1, we know there will be one multiplier in the solution space. When we reach node 2, there will be two cases: {1, 2} or {(1, 2)}. {1, 2} means that 1 and 2 occupy two different multipliers, and {(1, 2)} means that 1 and 2 share the same multiplier. Each case represents one solution point for the design processed so far. When we reach node 3, we know that there have to be two multipliers in the design because 2 and 3 are not compatible. The possible solution points will be {1, 2, 3}; {(1, 2), 3}; and {(1, 3), 2}. Similarly, we will process node 4, which will have a total of seven solution points. All of the solution points on node 4 inherit the solution points generated on node 3. In other words, solution points on node 3 propagate to node 4. For example, solution points {(1, 2), 3, 4}, {(1, 2), (3, 4)}, and {(1, 2, 4), 3} on node 4 all inherit solution point {(1, 2), 3} and search along three different directions by either letting 4 occupy its own multiplier or share an existing multiplier already in the datapath. Each solution has its power and delay values based on the datapath implemented according to the solution point.

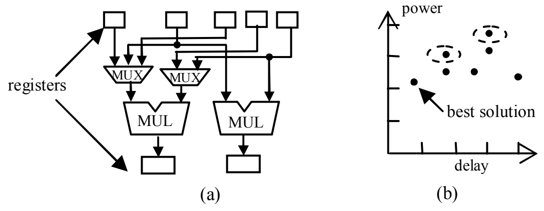

Figure 3: (a) datapath for {(1, 2, 4), 3}; (b) solution curve

Figure 3(a) is the datapath according to one of the solution points: {(1, 2, 4), 3}. We use the longest combinational path in Figure 3(a) as the delay for this solution point (a combinational path starts from one of the registers on top and ends at one of the registers on bottom). We use the estimated power value (Section 4) as the power for the solution point. Notice multiplexers are naturally included in the power and delay calculations. Figure 3(b) illustrates the curve of solution points when we finish operation 4. Notice some solution points are inferior (those in dashed ovals), i.e., they have the same delay as another solution point but with larger power. The inferior solution points will be pruned from the solution space. Only non-inferior solution points will be propagated to the next node.

It is intuitive that the accuracy of delay and power estimation is directly related to solution pruning and final solution quality. Counting all the contributing components accurately has a direct impact to reduce the total amount of multiplexing and interconnecting requirements for better global power and/or delay minimization. For any two solution points, suppose *Sol1* has *[power1, delay1]*, and *Sol2* has *[power2, delay2]*, we do not enforce that when *delay1* > *delay2*, *power1* has to be smaller than *power2*. This is because that the solution points represent the resource configurations in the partial datapaths before the end of the search, and the final desired datapath may be quite different. Therefore, we do not want to be too strict and greedy during the solution space search procedure. As long as two solution points have different delays, we keep them. Of course, there is an upper limit on the number of solution points we can keep. The more solution points, the larger solution space we are able to search but with larger runtime. We keep *M* solution points that possess the first *M* shortest delays explored so far. The far left point in Figure 3(b) represents the final best solution in terms of both power and delay among all the solutions. Different designs will have different curves. It is possible that a smaller power has to be achieved by sacrificing performance, or a smaller delay has to be achieved by sacrificing lower power. Due to space limit, we omit a formal description of the algorithm. We observe that a small number of solution points (e.g., $M = 10$) can already produce excellent results that are close to those generated through a larger number of solution points (e.g., $M = 50$). The runtime of the exploration is fast — usually within 1 minute with a 2GHz Linux machine.

6. EXPERIMENTAL RESULTS

6.1 Simulation and Power Estimation Analysis

To evaluate our high-level power estimator, we designed a verification process. After xPlore-Power generates its synthesis solution, it reports the estimated delay and power for the design. The test vectors used for our power estimation are dumped out in a format that Altera Quartus II can take. We then pass this vector file and our generated VHDL file containing our synthesis solution to Quartus II for RTL synthesis, placement and routing, timing analysis, and simulation. Afterwards, Quartus II's built-in power estimator *PowerPlay* power analyzer [1] will report the power consumption of this design based on its gate-level simulation result. We use test vectors that have very high simulation coverage (up to 96.7%).[2] Therefore, the final power reported from PowerPlay is quite accurate. xPlore-Power is incorporated into xPilot and uses the built-in data model and design flow from xPilot [29].

We present some detailed data in Table 4. The benchmarks are all in C and are a mixture of data-intensive (DFG) and control-intensive (CDFG) designs. Designs *dir, mcm, lee* and *pr* are DCT algorithms or DSP programs. Design *motion* is an algorithm to compute motion vectors, which is useful for video compression. Design *sym_conv* computes 2D DWT of a 128x128 image. We use the smallest chip, EP1S10B672C6, from the Stratix family. The reported power values include both dynamic and static power. Dynamic power includes power contributed by logic elements, DSP cores, I/O pins, clocks, and memories. The static power reported from PowerPlay is fixed for each device. It is 187.50 mW for device EP1S10B672C6. Therefore, we also use a fixed static power in our estimation. Overall, we can observe that the estimated power from xPlore-Power is very close to the power reported by PowerPlay after placement and routing, with an average error of 8.7% based on the absolute values of estimation errors. This indicates that our high-level power estimation is sound and effective.

Benchmarks	PowerPlay (mW)	xPlore-Power (mW)	Estimation Error (%)
dir	437.7	431	-1.5%
lee	1814.8	1533.4	-15.5%
mcm	390.7	423.4	8.4%
motion	239.3	252.1	5.3%
pr	1491.3	1536.7	3.0%
sym_ conv	307.2	251.4	-18.2%
	Absolute Value Average:		**8.7%**

Table 4: High-level power estimation compared to PowerPlay [1]

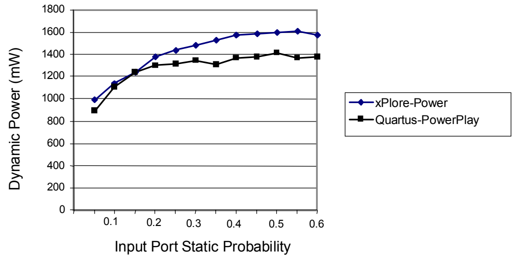

Figure 4: Estimated and reported power over static probability on benchmark *pr*

Figure 4 shows the correlation between the estimated power and reported power from another angle. The x-axis is the static probability (the probability of being logic high) on the input pins of the design. Different static probabilities on the input pins imply different switching activity on the inputs.[3] We observe that the two curves are very close and have a similar trend. This shows that our power estimator is sound and able to provide meaningful guidance for the low-power design space exploration.

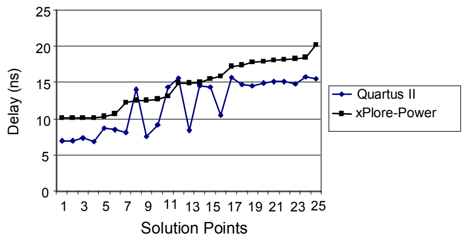

Figure 5: Estimated and reported delays with solution points for benchmark *motion*

To verify the fidelity of our delay model, we carry out another experiment to compare delays reported from both xPlore-Power and Quartus II. Figure 5 illustrates the details. In xPlore-Power, the solutions are sorted by an increasing order on estimated delay values. Therefore, the curve for xPlore-Power is increasing along the solution point number. We can observe that the delay values reported by Quartus II after placement and routing correlate to those of xPlore-Power, although there are some swings along the curve. The swings are expected because the xPlore-Power delay model can not predict the wire delay in the routing tracks without the actual layout. We can still observe that these two curves are very close, which indicates that our delay estimation is meaningful and sound.

6.2 xPlore-Power vs. Traditional Allocation and Binding Algorithm

To verify the effectiveness of our algorithm, we carry out a comparison study with a traditional allocation and binding algorithm

[2] A percentage from PowerPlay reporting the ratio of output ports actually toggling between 1 and 0 during simulation, compared to the total number of output ports present in the netlist.

[3] The switching activity for the input can be calculated by a formula as $2 \cdot Pv \cdot (1 - Pv)$, where Pv is the probability of input v being 1.

using graph coloring. We only examine register allocation and binding here to narrow down the comparison criteria. Our algorithm carries out functional unit and register allocation and binding through xPlore-Power while the graph coloring algorithm will carry out the same functional unit allocation and binding through xPlore-Power, but register allocation and binding through its own method. The goal of graph coloring algorithm is to minimize the number of registers during allocation.

Graph coloring is a well-known technique to solve binding and allocation problems. To color the lifetime conflict graph of the dataflows with the minimum number of colors is equivalent to finding the best clique partitioning solution on the corresponding compatibility graph of the same dataflows. Some previous work on high-level synthesis used similar algorithms [9][28]. We compare our result to that generated from a well-known package, *lmXRLF*, available from [17]. lmXRLF purely works on coloring the graph with the minimum number of colors. It designs a novel search algorithm to find the best independent set in the graph, one by one, according to an objective function, which is related to number of incident edges as well as two layers of neighborhoods. For our case, finding an independent set in the conflict graph is equivalent to finding a binding solution for all the nodes in the independent set (they are all compatible with one another). To make lmXRLF power aware, we estimate the switching activities for the nodes included in an independent set, and change the original cost by considering switching activity factors. We name this variant of lmXRLF as *lmXRLF-Power*.

Table 5 shows the detailed power and Fmax values for each algorithm. On average, lmXRLF-Power only offers a 3% improvement on power consumption compared to lmXRLF. The reason is that it only models the power consumption of the registers and does not model the multiplexers generated for the datapath. On the other hand, xPlore-Power is 32% better on power and 16% better on Fmax compared to lmXRLF. All the data are obtained after placement and routing using Quartus II.

	lmXRLF		**lmXRLF-Power**		**xPlore-Power**	
Bench marks	**Power (mW)**	**Fmax (MHz)**	**Power (mW)**	**Fmax (MHz)**	**Power (mW)**	**Fmax (MHz)**
dir	541.9	160.1	447.7	153.7	250.2	236.3
lee	3955.6	113.6	4129	107.9	1627	122.9
mcm	492.9	171.9	500.9	174.6	203.2	241.1
motion	56.5	139.3	56.6	145.6	51.8	142.1
pr	1418.8	114.2	1360.5	111.0	1304	111.3
sym_conv	155	71.2	155	71.2	146.5	73.7

Table 5: xPlore-Power vs. lmXRLF (a graph coloring algorithm)

7. CONCLUSIONS

In this paper we concentrated on resource allocation and binding tasks to optimize FPGA power and delay. We designed a high-level power estimator for a commercial FPGA architecture. We proposed a new simultaneous allocation and binding optimization algorithm, xPlore-Power, for efficient design space exploration. We handle all the contributing resources in the datapath. Our high-level power estimator is only 8.7% away from a commercial gate-level FPGA power estimator. Comparing to a traditional graph coloring-based register binding algorithm, xPlore-Power is 32% better on power and 16% better on Fmax after placement and routing.

Acknowledgements

This work is partially sponsored by NSF under grant CCF-0306682 and Altera, Magma, and Xilinx under the MICRO Program.

REFERENCES

[1] Altera Corp., PowerPlay Power Analyzer, *http://www.altera.com/support/devices/estimator/pow-powerplay.html.*

[2] Altera Corp., Stratix Device Handbook, *http://www.altera.com/literature/hb/stx/stratix_handbook.pdf.*

[3] Altera Corp., Stratix PowerPlay Early Power Estimator, *http://www.altera.com/support/devices/estimator/powpowerplay.html.*

[4] A. Bogliolo, et. al, "Efficient Switching Activity Computation During High-Level Synthesis of Control-Dominated Designs," *ISLPED*, Aug. 1999.

[5] J. M. Chang and M. Pedram, "Register Allocation and Binding for Low Power," *Design Automation Conf.*, 1995.

[6] D. Chen and J. Cong, "Register Binding and Port Assignment for Multiplexer Optimization," *ASPDAC,* Jan. 2004.

[7] D. Chen, J. Cong, and Y. Fan, "Low-Power High-Level Synthesis for FPGA Architectures," *Int. Symp. Low Power Elec. and Design*, Aug. 2003.

[8] K. Choi and S. Levitan, "A Flexible Datapath Allocation Method for Architectural Synthesis," *ACM TODAES*, Vol. 4, No. 4, Oct. 1999.

[9] J. Cong, et. al, "Bitwidth-Aware Scheduling and Binding in High-Level Synthesis," *Asia South Pacific Design Automation Conf.*, Jan. 2005.

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

[11] D. Gajski et. al., Editors, High-Level Synthesis – Introduction to Chip and System Design, Kulwer Academic Publishers, 1992.

[12] CH Gebotys, "Low Energy Memory and Register Allocation Using Network Flow," Design Automation Conf., 1997.

[13] CH Gebotys and MI Elmasry, "Optimal Synthesis of High- Performance Architectures," IEEE J. of Solid State Circuits, 27, 3, 389-397, 1992.

[14] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits, Springer, 2004.

[15] C.Y. Huang, et. al, "Data Path Allocation Based on Bipartite Weighted Matching," Design Automation Conf., 1990.

[16] T. Kim and C.L. Liu, "An Integrated Data Path Synthesis Algorithm Based on Network Flow Method," IEEE Custom Integrated Circuits Conf., 1995.

[17] D. Kirovski and M. Potkonjak, "Efficient Coloring of a Large Spectrum of Graphs," Design Automation Conf., June 1998.

[18] P. Kollig and B. M. Al-Hashimi, "Simultaneous Scheduling, Allocation and Binding in High Level Synthesis," Electronics Letters, vol. 33, 1997.

[19] E. Kusse and J. Rabaey, "Low-Energy Embedded FPGA Structures," Int. Symp. on Low Power Electronics and Design, Aug. 1998.

[20] F. Li, D. Chen, L. He, and J. Cong, "Architecture Evaluation for Power-efficient FPGAs," Int. Symp. on FPGA, 2003.

[21] T. A. Ly and J. T. Mowchenko, "Applying Simulated Evolution to High Level Synthesis," IEEE Tran. on CAD, Vol. 12, No. 3, Mar. 1993.

[22] S. Ogrenci Memik, G. Memik, R. Jafari, and E. Kursun, "Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs," Design Automation Conf., 2003.

[23] A. Mujumdar, R. Jain, and K. Saluja, "Incorporating Performance and Testability Constraints during Binding in High-Level Synthesis," IEEE Tran. on CAD, Vol. 15, no. 10, Oct. 1996.

[24] B. Pangrle, "On the Complexity of Connectivity Binding," IEEE Tran. on CAD, Vol. 10, no. 11, Nov. 1991.

[25] S. Raje and R. A. Bergamaschi, "Generalized Resource Sharing," Int. Conf. on Computer-Aided Design, Nov. 1997.

[26] M. Rim, R. Jain, and R. De Leone, "Optimal Allocation and Binding in High-level Synthesis," Design Automation Conf., 1992.

[27] A. Singh and M. Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," Int. Symp. on FPGA, Feb. 2002.

[28] C-J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Path in Digital Systems," IEEE Tran. on CAD, Vol. CADJ, No.3, Jul. 1986.

[29] xPilot: Platform-based Behavior Synthesis System, *http://cadlab.cs.ucla.edu/soc/index.htm.*

[30] Y. Zhang, X. Hu, and D. Z. Chen, "Efficient Global Register Allocation for Minimizing Energy Consumption," SIGPLAN Notices, 37(4): 42-53, 2002.