

Mapping for Better Than Worst-Case Delays In LUT-Based FPGA Designs

Jason Cong and Kirill Minkovich
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA
{cong, cory_m}@cs.ucla.edu

ABSTRACT

Current advances in chip design and manufacturing have allowed IC manufacturing to approach the nanometer range. As the feature size scales down, greater variability is experienced, forcing designers to reduce performance requirements in order to reserve larger margins. Better than worst-case design can be used to address the variability problem, as well as breaking the performance limit set by the worst-case delay in the conventional design style, even without the consideration of delay variation. In this paper we will present a novel methodology for measuring and optimizing the performance of circuits to operate with the clock period smaller than the worst-case delay. We also develop a novel technology mapping algorithm that optimizes circuits under such a metric. Using our novel mapping algorithm named BTWMap (Better Than Worst-case Mapper) and its area-optimized version named BTWMap+area, we are able to improve the overall circuit latency by 13% and 11%, respectively.

Categories and Subject Descriptors

B.6.3 [Hardware]: Logic Design – *Optimization*

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Logic Synthesis, Technology Mapping, FPGA Lookup Table, Razor, Better Than Worst-Case, Simulation, Switching Probabilities

1. INTRODUCTION

As transistor sizes scale the down to the nanometer range, many new design challenges occur, and delay variation is one that may significantly limit the circuit performance. Delay variation is mainly caused by increasing environmental and process variation. Environmental variation can be caused by overheating and voltage fluctuations, while process variation is caused by dopant density variation, edge geometry variation, and stress that may occur during manufacturing. Even at 65-nanometers, device sizes are well below the wavelength of the light used to pattern them, resulting in considerable edge geometry variation. As device sizes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA '08, February 24–26, 2008, Monterey, California, USA.

Copyright 2008 ACM 978-1-59593-934-0/08/02...\$5.00.

further shrink, the problems become greatly amplified. In the conventional design style considering only worst-case delays, delay variations have usually been tolerated by adding enough margin or slack to allow for a 3-sigma delay variation or higher. Although adding such a margin allows many more chips to meet their timing constraints, these timing constraints themselves are conservative. To improve profit margins and performance, CPU and FPGA manufactures speed bin their products by selling different speed grades of the same chip. But most ASIC vendors don't have the luxury of selling different versions of the same product and in turn suffer from the significant losses in potential circuit performance caused by their conservative models.

Recently, a number of research efforts have been made to improve circuit performance through statistical optimization or variation-aware optimization at the circuit-level or layout-level through cell sizing [3][4] or buffering [15]. The objective of these works is to improve the probability density function (PDF) of the circuit delay. While these optimizations are helpful in improving the timing yield, they still assume that circuit performance is determined by the worst-case delay.

There is a common consensus that circuit variation is going to get much worse as we approach the limit of CMOS scaling and start exploring different nanotechnologies as alternatives to CMOS scaling. We believe that a fundamental approach to dealing with design uncertainty is an architectural design innovation which would allow us to design and optimize circuits that can operate with better (smaller) than worst-case delay. (Another approach will be a complete asynchronous design, but it requires significant changes to the exiting design flow and design tools, which we shall not discuss in this paper.) In fact, the better-than-worst-case delay methodology may improve circuit performance *even in the absence of process variation* (as we shall show later in this paper), as many circuits rarely activate the worst-case timing path (such as the long carry chain in a 64-bit adder). For example, when examining the switching activity of the mapped MCNC benchmark suite, we have some interesting observations -- 75% of the circuits had most of their switching activity occur earlier than the circuit depth (i.e. the longest path). This earlier switching activity means that 55% of the un-optimized circuits would see an immediate performance benefit (decrease in expected delay) after being implemented in a better than worst-case architecture. Given these considerations, there is a lot of potential behind better than worst-case designs.

In this paper, we develop the performance optimization metric and efficient technology mapping algorithms for general synchronous designs that can operate at better than the worst-case delay. Our architecture is based on the Razor architecture [1]

introduced recently to overcome both design uncertainty and architecture limitations. The Razor architecture was originally developed for microprocessors with the well-defined pipeline architecture. It allows circuits to be clocked faster than the worst-case delay by having an extra set of registers clocked with a delayed clock. Using these registers, the Razor architecture can tell if one of the registers latch a data which is not stabilized during the given clock period, signal an error, and stall the pipeline. The exact delay error handling pipeline stage (originally presented in [1]) is shown in Figure 1. The shadow latch is clocked with the delayed clock, which allows it to catch any errors caused by the main flip-flop being clocked early. When an error is detected (the shadow latch and the main flip-flop have different values), an error signal is generated. This error signal allows the main flip-flop to copy the value from the shadow latch while it stalls all the pipeline stages. During this time, the next logic stage has time to recalculate its values using the correct inputs.

Originally, the Razor architecture was designed for a pipelined microprocessor architecture in order to reduce the power consumption via dynamic voltage scaling. But the underlying concept can also be applied to synchronous circuits controlled by a FSM (finite state machine) to improve the circuit performance. This is possible by adding the proper clock gating signals to stall the FSM for an extra clock cycle whenever the main flip-flop and the shadow register disagree.

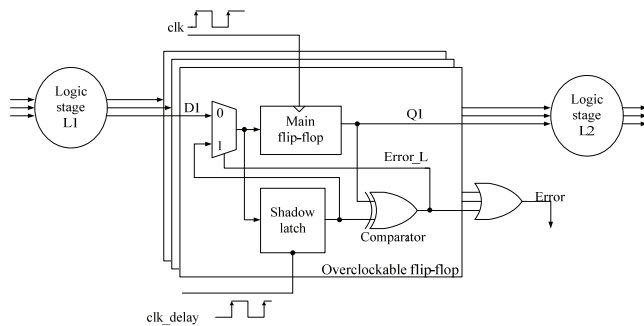


Figure 1. Razor stage with shadow latches and control lines.

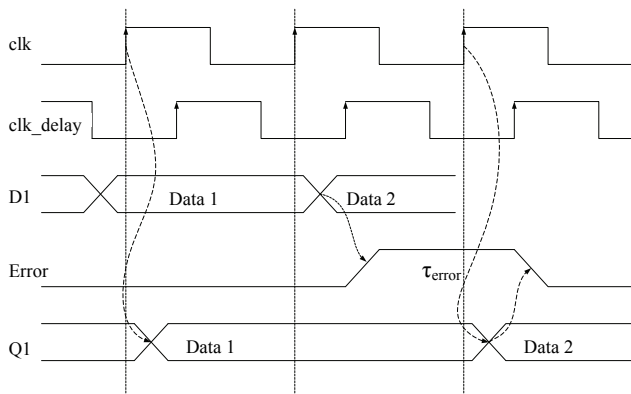


Figure 2. Timing diagram for Razor control lines.

Let us examine Figure 2, a detailed timing diagram (which was originally presented in [12]), to get a better understanding of how the Razor architecture can recover from error. In this example, after the second clock cycle, the main flip-flop latches Data 2 too early, while the shadow latch, using the delayed clock, receives

the correct data. Immediately, the error signal goes high (for every latch) causing the next cycle to move the shadow latch's value to the main flip-flop. At the same time, the error signal stalls the transition of the FSM, so that an extra clock cycle can allow the shadow register to load the correct values into the regular registers. Thus, any timing error costs one extra clock cycle. We call such an architecture *stallable FSM architecture*. Figure 3 shows how to transform a standard FSM architecture into a stallable FSM architecture. This can be done by converting all the registers to their equivalent versions with shadow registers as in Figure 1 (labeled as “Razor registers” in Figure 3), adding a Razor stabilization stage, and allowing the inputs to be buffered and stalled by the timing error signal. In this translation, it is important to note two things. First, it is not necessary to replace all the registers with their Razor counterparts, but just the ones that might produce errors. Second, the more registers converted to Razor, the faster we can potentially clock the circuit. However, there is a limit to how fast we can clock the circuit. By examining Figure 2, we can see that if we increase the clock frequency by 2X or more (compared to the worst-case delay clock) then the *clk_delay* may not catch all the errors. This is because in order to catch all the errors, the time between the rising edge of *clk* and the next rising edge of *clk_delay* has to be less than the worst case delay.

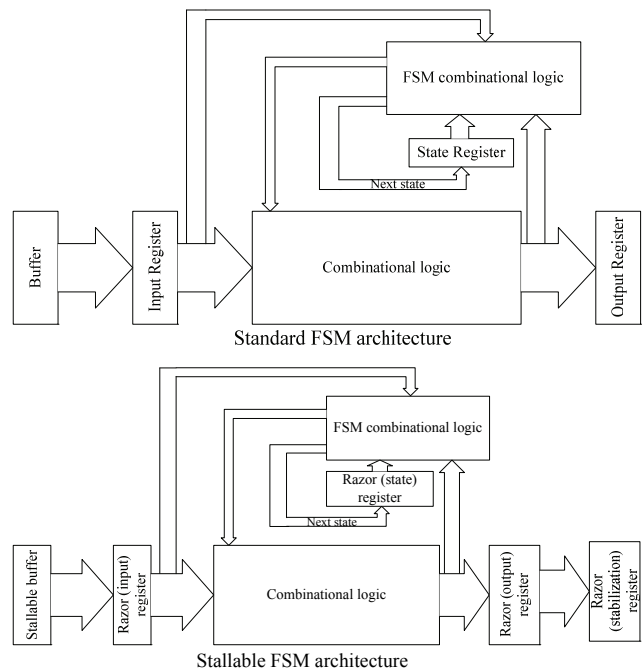


Figure 3. Conversion to the stallable FSM architecture.

In standard logic synthesis for the worst-case delay minimization, mapping for the minimum depth while optimizing area usually leads to good delay on a placed and routed solution. However, such optimizations may not be as useful for the stallable FSM architectures presented here, since their operating frequency is not determined by the worst-case path delay, but is instead more closely correlated to the typical-case delay. To address such new types of architectures, we have developed a new technology mapping tool called Better Than Worst-case Mapper (BTWMap). A natural question is if implementing a stallable FSM architecture will provide enough performance benefits to outweigh the extra

area and design complexity. This question was partially answered in [12], where a Razor processor was implemented in ASIC without any special logic optimization, and was still able to achieve impressive improvements (in power savings). It is likely that if they actually performed the special logic optimization targeted for the better than worst-case delay, as done in this paper, they would have seen even better gains. We believe that stallable FSM architectures can achieve similar area to performance tradeoffs as aggressive pipelining, since both methods increase the area and complexity in hopes of better performance. The demonstration of the stallable FSM architecture on FPGAs has not been done. But we plan to do such a study soon, using the BTWMapper in this paper and other logic synthesis tools under the development in our lab targeted for the better than worst-case delay optimization.

In the remainder of this paper we will present a method for measuring the expected delay for circuits optimized for better than the worst-case delay. We present the BTWMap algorithm, which is able to reduce the expected delay by 13% but with 26% area increase. We will also present an enhanced version of BTWMap algorithm with area recovery called BTWMap+area, which only increases the area by 10%, but still maintains a reduction of expected delay by 11%. This was made possible by performing area/performance trade-offs, as well as supplying the algorithm with a target clock period, which allowed the algorithm to ignore any transitions that happened before this delay.

2. DEFINITIONS AND PROBLEM FORMULATION

To understand the complex notion of measuring delay on a delay error-resistant architecture, one first has to understand a simpler notion of propagation delay. For simplicity, as in most of other technology mapping works [6][8], we use the unit delay model (prior to placement and routing). However, our technique can be extended to post-placement optimization with more accurate gate delay models. Under the unit delay model, for a particular input sequence, the *propagation delay* is the maximum level for any gate output to transition to its final value under the given input sequence. Given the previous input $i_p=[i_1, i_2, \dots, i_n]$ and the current input $i_c=[i'_1, i'_2, \dots, i'_n]$, the transition gate delay, $d(i_p, i_c)$, is the number of gate delays the circuit f takes to go from state $f(i_p)$ to $f(i_c)$. For example, when one considers Figure 4, going from input $[1,1,1,1]$ to $[1,1,1,0]$ takes two gate delays, while the transition from $[1,1,1,0]$ to $[1,1,0,0]$ takes zero gate delays, since there is no signal transition at the output f .

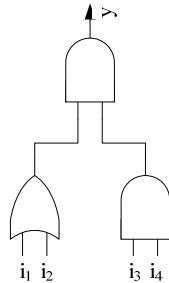


Figure 4. An example showing transition delay differences.

For conventional architectures, the worst-case delay, or the longest path, is all that is really needed, and this can be easily calculated in linear time using a simple topological traversal. The stallable FSM architecture, on the other hand, requires a much more complex calculation. The original paper on the Razor architecture [1] presented the simple notion of measuring performance by measuring the *typical-case delay*. Typical-case delay is the average time it takes an input sequence to propagate to the outputs. The problem is that it does not accurately predict the performance of the circuit. Since our stallable FSM implementation only slows down when it sees an error, the performance is determined by the likelihood of a timing error, and the typical-case delay can not be used to calculate this. For this reason, we introduce *the expected delay*, to be defined below, as a better metric for measuring the performance of a stallable FSM under a given input sequence.

2.1. The Expected Delay

The expected delay determines the performance of the circuit that will be implemented in an error recovery scheme. It can easily be computed by dividing the total processing time by the amount of data processed, just like in processor throughput computations. This calculation is needed since different input sequences can take varying amounts of time to propagate to the outputs. Assuming a unit delay model, let P_{wrong}^d be the probability of circuit switching at depth d or higher. Then the expected delay of a stallable FSM with the target clock period d can be computed using the simplified formula (2) below, since τ_{error} , the time to detect and recover from an error, is less than d (see Figure 2).

$$P_{wrong}^d = \frac{\# \text{ input vectors that cause transitions after } d}{\# \text{ input vectors}} \quad (1)$$

$$\begin{aligned} ExpDelay(d) &= d \cdot (1 - P_{wrong}^d) + (d + \tau_{error}) \cdot (P_{wrong}^d) \\ &\leq d \cdot (1 - P_{wrong}^d) + (d + d) \cdot (P_{wrong}^d) \\ &\leq d \cdot (1 + P_{wrong}^d) \end{aligned} \quad (2)$$

Formula (2) expresses the expected delay for a specific target clock period d , but to find the best expected delay (and its corresponding target clock period), a simple linear search (3) can be done over all the possible target clock periods. In order to guarantee the correctness, the clock cannot be increased by 2X or higher of the one determined by the worst-case delay. Thus, the search starts at half the maximum depth.

$$BestExpDelay = \min_{d=\frac{\max_depth}{2}}^{\max_depth-1} ExpDelay(d) \quad (3)$$

Because we are using a unit delay model, transitions can only happen at discrete times in the clock period. We keep track of the switching probabilities at different levels by keeping a vector for each node called the *switching probability vector*.

To calculate the gate switching probability exactly is an NP-hard problem for general circuits, because SAT can be reduced to checking if the probability of the output transitioning to one at the last depth is greater than zero. One way to reduce the complexity is to use simulation to calculate these values. To calculate the switching probability exactly would require a full simulation, but that would be time prohibitive, since all pairs of inputs have to be simulated. For a circuit with n inputs, it would take $O(2^{2n})$. In our

approach, we use partial simulation for run time and accuracy trade-off. That is, we use the simulation-based solution to compute the switching probability under a given sample input sequence of significant length (say 100,000 vectors).

2.2. Problem Formulation

In this paper we will consider a method for improving the performance of circuits implemented using stallable FSM architectures on field programmable gate arrays (FPGAs). FPGAs have been gaining momentum as an alternative to application-specific integrated circuits (ASICs). FPGAs consist of programmable logic, I/O, and routing elements which can be programmed and reprogrammed in the field to customize an FPGA, enabling it to implement a given application in a matter of seconds or milliseconds. The most common type of programmable logic element used in an FPGA is called a K -LUT, which is a K -input 1-output lookup table (LUT) capable of implementing any K -input 1-output Boolean function. K -LUTs are commonly implemented as a 2^K truth table on the LUT's inputs. Technology mapping is a step in logic synthesis that maps an arbitrary Boolean network to a network of LUTs. The goal of traditional delay-oriented technology mapping is to minimize the number of LUTs needed to cover all the logical elements in the circuit while achieving the optimal mapping depth.

The goal of the better than worst-case mapping formulation in this paper is to minimize the expected delay (defined in Section 2.1) while keeping any area increase to a minimum. In an ideal situation, first the optimal expected delay is calculated, from which the target clock period can be calculated. Then, using the target clock period, some of the area can be recovered. However, since we are unable to calculate the optimal expected delay directly, we perform a linear search and compute the mapping solution with the best expected delay for each given target clock period. We then perform aggressive area reductions under this target clock period. Finally, we choose the best target clock period that leads to the smallest expected delay and report the corresponding mapping solution.

3. BTWMap

BTWMap was created to optimize the logic for an FPGA using the cut-based approach, which was first used in ZMap [8] and later adopted in DAOMap [6], IMap [13], and ABC [16]. To perform this type of logic optimization, a set of cuts, or possible implementations, is first generated for each node. Then, the algorithm assigns a cost to each cut and recursively selects the best cut in an output-to-input order. The exact details of our implementation will be described in Section 3.1.

When working with switching probabilities, the difficulty for any algorithm can be attributed to two obstacles. First, switching probabilities are hard to compute. Second, it is possible that the switching probability vectors of many gates may change after some gates are mapped.

To determine the switching probability of a gate, it is necessary to look at three things: the function of the gate (is it an AND, XOR, ... etc.), the switching probabilities of the inputs, and any "don't cares" (impossible inputs) that arise from reconvergent wires. The main problem arises from the fact that the last two requirements are difficult to calculate. Full don't cares generation for every node is known to be very expensive in terms of runtimes and memory requirements [11][14]. Calculating the exact switching

probabilities of inputs is very difficult. To overcome these difficulties, BTWMap uses a simulation-based method with several custom speedups.

The BTWMap algorithm, based on the cut-enumeration framework [9][16], will be presented in two parts. First, we will present our cost function and describe how it is used (Section 3.1). Then we will present several area/performance tradeoffs specifically designed for better than worst-case architectures (Section 3.2).

3.1. Cut Selection

The cut selection process, described in Figure 6, is an iterative process in which the outer loop iterates over two parts. First, several input sequences are simulated over the mapped network and switching probability vectors are generated for each cut (lines 7-24). Second, using these vectors, costs can be generated and new cuts can be chosen (lines 25-30) to form the new mapped network for the next iteration.

3.1.1. Cost Function for a Cut

The critical part of every cut-based synthesis algorithm is selecting the best cut, and BTWMap is no different. Selecting a good cut is difficult because so many factors may influence which cut is the best. Our main goal is to select a cut that allows the node to transition to its final value as quickly as possible. In the BTWMap algorithm, every cut has a vector of switching probabilities (the probability that it will switch at a certain delay) which helps the algorithm determine which cut is the best. The d^{th} position in the switching probability vector represents the probability that the function will change values between time d and $d+1$. For example, consider the example in Figure 4 and consider mapping that circuit to a 3-LUT architecture. The two possible cuts of node y are shown below in Figure 5 and their corresponding switching probability vectors are shown in Table 1. By looking at the switching probabilities in Table 1, it is clear that Cut 2 is the better choice. This is due to the fact that the output signal y is less sensitive to signal p than signal q , where switching sensitivity is defined in Formula (4) below.

$$\text{switching_sensitivity}(y, x_i) = \Pr(y \text{ changes} \mid x_i \text{ changed}) \quad (4)$$

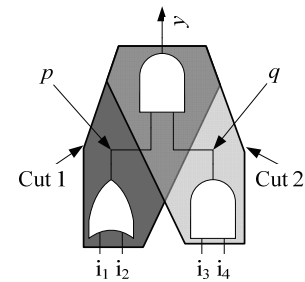


Figure 5. An example that demonstrates cut choosing.

Table 1. Example of cut choosing for Figure 5.

Depth	Probability of switching	
	Cut 1	Cut 2
2	28%	9%
1	9%	28%

In general, it is not always clear (as in the previous example shown in Table 1) which cut is better. For example, considering the case shown in Table 2 below, it is clear that even though Cut 1 has a better switching probability at depth 3, choosing Cut 2 would probably be a better choice. This is because Cut 2 has much better switching at depth 2, and this will affect the switching probabilities of its fanout nodes.

Table 2. Example of difficult cut choosing.

Depth	Probability of switching	
	Cut 1	Cut 2
3	3%	4%
2	50%	5%
1	70%	70%

To simplify cut selection, we introduce a single cost function (a depth-weighted summation over the switching probability vector) for each cut C based on the heuristic function below.

$$\text{cost}(C) = \sum_{\text{depth } d=0}^{\text{max_depth}} (d)^\alpha \cdot \text{switching_probability}(C, d) \quad (5)$$

It is possible, by changing α , to specify exactly how important a transition is at higher depth. If α is 0, the depth field d is ignored, and the switching activity is minimized. As α increases, so does the cost of switching at higher depths.

3.1.2. Cost Function Calculation and Propagation

In the formula above, the only thing that needs to be calculated and kept updated is the switching probability. To calculate the switching probability, we use a simulation-based method. The general idea is to first simulate a set of random input vectors (or a vector sequence given by a user based on realistic input distribution), then propagate the data over the nodes, select the best cuts, and repeat.

We implemented our BTWMap algorithm on top of Berkeley's ABC [16] mapper. At the start, ABC decomposes the circuit into a two input AND-INV circuit and performs cut enumeration on this modified circuit without pruning any of the cuts. After that, each cut is comprised of just two pointers to the two sub-cuts, left and right, and their corresponding phases (negated or not). This allows a simulation of 32 (the size of an integer) input vectors on an arbitrarily sized cut with just one AND and one NOT Boolean operation on average (since the calculation just looks up the simulation data of the left cut and the right cut, possibly performs some negation, and AND the two integers together). This enables BTWMap to compute the switching probability of all the cuts quickly and efficiently.

a problem occurs when the cut that was used to implement a node changes, since this can affect every switching probability of its fanout nodes. To have accurate switching probabilities, the nodes affected by this cut choice should be re-simulated, but that would take too much time. Instead, we opt for a more efficient heuristic that seems to work well in practice. First, BTWMap simulates some of the input sequence and chooses cuts for the whole circuit but instead of throwing away that partial simulation data, it assumes the old data still has some importance and stores them as a switching probability vector for each node. For the next iteration of cut selection, BTWMap uses both the simulation data from the

current iteration, $\text{cost}(C)$, and the old simulation data C_{cost}^{i-1} to generate the current iteration's cost function C_{cost}^i (formula (6)). At each iteration only 256 values are simulated, which is far too little to have meaningful switching activity, so we are forced to use old simulation data. The idea is that if enough iterations are performed and the last iterations see very few changes, then this cost function can be used to accurately compare two nodes.

$$C_{\text{cost}}^i = \beta \cdot C_{\text{cost}}^{i-1} + \text{cost}(C) \quad (6)$$

Varying β , we are able to control how much importance is placed on the results from the previous iterations of the simulation. As β increases to one, more importance is put on the previous iteration's switching activity calculation. In essence, BTWMap is trying to converge to the optimal solution by using the previous solutions as a starting point. Putting this all together, we get the simulatedCutSelection method described in Figure 6.

```

algorithm simulatedCutSelection (C)
input: the circuit C, output: mapped network G
1 foreach iteration
2   set  $S_{\text{curr}} = \emptyset$ ; //set of cuts for current iteration
3   set  $S_{\text{next}} = \emptyset$ ; //set of cuts for next iteration
4   foreach cut  $c$ 
5      $c.\text{cost} = 0$ ;
6   end-foreach;
7   foreach  $n \in \text{PIs}$ 
8      $n.\text{simData} = \text{random}()$ ;
9     foreach cut  $c$  s.t.  $n \in c.\text{cutset}$ 
10       $S_{\text{curr}} = S_{\text{curr}} \cup c$ ;
11    end-foreach;
12  end-foreach;
13   $\text{depth} = 0$ ;
14  while  $S_{\text{curr}} \neq \emptyset$  do
15     $\text{depth}++$ ;
16     $\text{sort\_unique}(S_{\text{curr}})$ ;
17    while  $S_{\text{curr}} \neq \emptyset$ 
18       $c = \text{pop}(S_{\text{curr}})$ ;
19       $\text{simulate}(c)$ ;
20      foreach cut  $c'$  s.t.  $c.\text{root} \in c'.\text{cutset}$ 
21         $S_{\text{next}} = S_{\text{next}} \cup c'$ ;
22      end-foreach;
23    end-foreach;
24     $S_{\text{curr}} = S_{\text{next}}$ ;
25  end-while;
26  foreach node  $n$ 
27    foreach cut  $c \in n$ 
28       $\text{calc\_cost}(c)$ ;
29    end-foreach;
30     $n.\text{LUT} = \text{pick\_cut}(c)$ ;
31  end-foreach;
end-for;

```

Figure 6. Simulation-based cut selection (BTWMap).

This method can be broken down into four parts, which are: initialization (line 2-6), supplying sampling data to the inputs (lines 7-12), simulation (lines 14-24), and cut selection (lines 25-30). During the whole process we use two sets, S_{curr} and S_{next} . S_{curr} holds the nodes that are going to be simulated

during the current iteration while S_{next} holds nodes for the next iteration. To further explain the algorithm, both the *calc_cost* and the *pick_cut* methods will be described in more detail. For the *calc_cost* method, BTWMap approximates cut C 's switching probabilities for depth d in formula (5) using formula (7) below, then uses it in formula (6). The idea is to see how many bits changed between the two simulation vectors.

$$\text{switching_probability}(C, d) = \frac{\text{bit_count}(\text{XOR}(C.\text{simData}[d], C.\text{simData}[d-1]))}{2^d} \quad (7)$$

The *pick_cut* method picks the cut with the smallest cost that does not increase the minimum mapping depth. The depth constraint is needed so that if BTWMap's solution is not enough of an improvement, it will still have the same depth as the original circuit. Therefore, it can do no worse than the original algorithm in terms of depth. Removing this depth constraint improves the switching activity at each depth for all the circuits and the best expected delay for some circuits. But there are some instances where the depth increase significantly diminishes the best expected delay performance. The worst-case depth places an upper bound on the target clock period in our stallable FSM architecture, because latching the main flip-flop faster than twice the shadow latch clock (the target clock period) would lead to the shadow latch erroneously latching data for the next cycle.

3.2. Area Recovery

3.2.1. Relaxation for Target Clock Period

During the normal BTWMap operation, the main goal is to reduce the switching activity of each node; a 26% area increase is thus experienced. To be able to recover some of this area, we present an area-efficient version of the mapper, named BTWMap+area, which enables better trade-off between the area and the expected delay. BTWMap+area considers both the target clock period and the maximum mapping depth (which can be computed easily, say using FlowMap [7], or in our cut enumeration framework [9]). For example, if the circuit has a maximum depth of 10 and a target clock period of 7 is specified, the algorithm will try to make sure the outputs will not change after 7 levels, while preserving the maximum mapping depth of 10.

Given a target clock period, d , BTWMap+area ignores all the switching that happens before d and calculates the switching cost starting from d (instead of 0) in Equation (5). This greatly increases the number of cuts the area recovery engine is allowed to examine, leading to a significant reduction in area. To pass this information to each node, we use the *switching threshold*. The switching threshold of a node n is a number d_n which allows the node to ignore all the switching before time d_n . The difference between the switching threshold of a node and the target clock period of a circuit is analogous to the difference between the required time of a node and the maximum depth of a circuit. The switching threshold calculation and propagation is exactly like the required time calculation and propagation used in [6].

3.2.2. Area/Performance Tradeoff

To further reduce some of the area, some of the performance was traded in exchange for an area reduction. While traversing the circuit, the algorithm reduces the area by emulating ABC's three-phase area recovery techniques [16] with several added restrictions. Each cut has to meet the timing constraints as well as not relax the node's switching cost by more than certain amount. To quantify this relaxation we will introduce the variable I_n which

will represent node n 's allowable increase (in terms of switching probability cost). An interesting question is how to assign I_n for each node n , since having a uniform I_n would mean each node has an equal effect on the outputs, which is not true. For example, consider the circuit below (Figure 7) with random inputs, signal y is much more sensitive to x_1 compared to the other two signals. Specifically, $\text{switching_sensitivity}(y, x_1) = .75$, while $\text{switching_sensitivity}(y, x_2) = \text{switching_sensitivity}(y, x_3) = 0.50$.

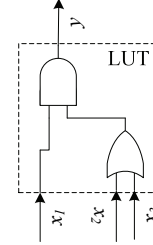


Figure 7. A LUT demonstrating differing correlations of input switching to output switching.

Using this information, it can be said that x_2 and x_3 can change more than x_1 because even if they change more often, the effect is not as great as x_1 changing (y is more sensitive to x_1 than x_2 and x_3). With these probabilities, we can determine exactly how much each input's I_n will be. Initially, each output node's I_n is set to γ and every other node's I_n is set to 0. Then, in a reverse topological order from POs to PIs, each node chooses the best cut within the constraints, propagating a proportional amount of slack I_n to its children. The exact amount propagated to the children is directly proportional to how that child will affect the current node's switching. Assuming the function is y and the input is x_i then Equation (8) can be used to calculate I_{x_i} .

$$I_{x_i} = \underbrace{(I_y.\text{given} - I_y.\text{used})}_{I_y.\text{slack}} \cdot (1 - \text{switching_sensitivity}(y, x_i)) \quad (8)$$

When considering 4-LUTs we can enumerate all possible functions (2^{16} or 65,536) and since there are only 256 possible transitions (2^4 before and 2^4 after) we can store the sensitivity information (number of output changes when input changes) for each input using just one integer (four bytes). This allows this whole table to be stored using only 256KB which we pre-computed and stored as a file to reduce the run time.

To summarize, BTWMap minimizes probability of getting an error while assuming the circuit is going to be clocked at the maximum possible clock speed (2x the worst-case delay). It then allows the user to assign a target clock period which is used, in conjunction with area/performance tradeoffs, to significantly minimize the area.

4. RESULTS

Our results will be presented in four subsections. First, we will discuss which circuits are acceptable for the BTWmap algorithm and which ones are not. Second, we will examine the switching activity to get a better understanding of what BTWMap is doing. Third, we will discuss how our set of algorithms improves the performance of circuits implemented using a better than worst-case architecture. Fourth, we will show BTWMap's area/expected-delay trade-off.

4.1. Benchmarks

The 20 largest MCNC circuits all fall into three categories. Let P_{wrong}^d be defined as in Section 2.1. First, there are the circuits that have no need to be optimized as they have a $P_{wrong}^{(\max \text{ depth})/2}$ of less than 2% even with the traditional depth-optimal mapping algorithm as such as the FlowMap [7] and ABC mapper [16]. There is no benefit to optimize these circuits any further since they can already be clocked at the half of the maximum mapping depth (which is the fastest target clock period that the stallable FSM architecture may operate), and they account for four of the MCNC suite (diffeq, elliptic, frisc and tseng). Second, there are the circuits that are hard to optimize which have a $P_{wrong}^{(\max \text{ depth})-1}$ of more than 90%. These circuits account for five of the MCNC suite (bigkey, des, dsip, ex1010 and ex5p) and it will not make much sense to implement them using a stallable FSM architecture. The third category accounts for the remaining circuits, which are meaningful candidates to benefit from the BTWMap algorithm. In general, to test to see if a circuit qualifies for optimization by the BTWmap under the stallable FSM architecture, one can just map it using ABC’s FPGA mapper and check the switching activity to see if it falls into the first two categories. Since these circuits can easily be tested and disqualified, they are not included in our benchmarking.

4.2. Switching Activity Reduction

In this section we will examine how BTWMap and BTWMap+area differ from Berkeley’s ABC mapper [16] in terms of switching activity. In our experiment, we mapped them into 4-LUTs using these different methods. For the ABC results, we used the “-K 4” option to map to 4-LUTs. To evaluate any gains, we created our own logic and timing simulator. The timing simulator, based on a unit delay model, reports back the likelihood of any output changing after a certain amount of time. This global output switching activity allows us to accurately model how this circuit will perform under the stallable FSM architecture. A good way to understand what BTWMap tries to do is by looking at the global output switching activities in the mapping solutions of MCNC example PDC as shown in Figure 8 below. The figure shows that both versions of BTWMap significantly decrease the probability that any output will switch, especially for the last depths. Initially, BTWMap tries to reduce the switching activity at every depth, then the area techniques recover some of the area by realizing that only the switching at the larger depths is important (depth 7 and beyond for the example below). As you can see, both versions of BTWMap stay very close to each other from delay 9 to 7, where they start to diverge.

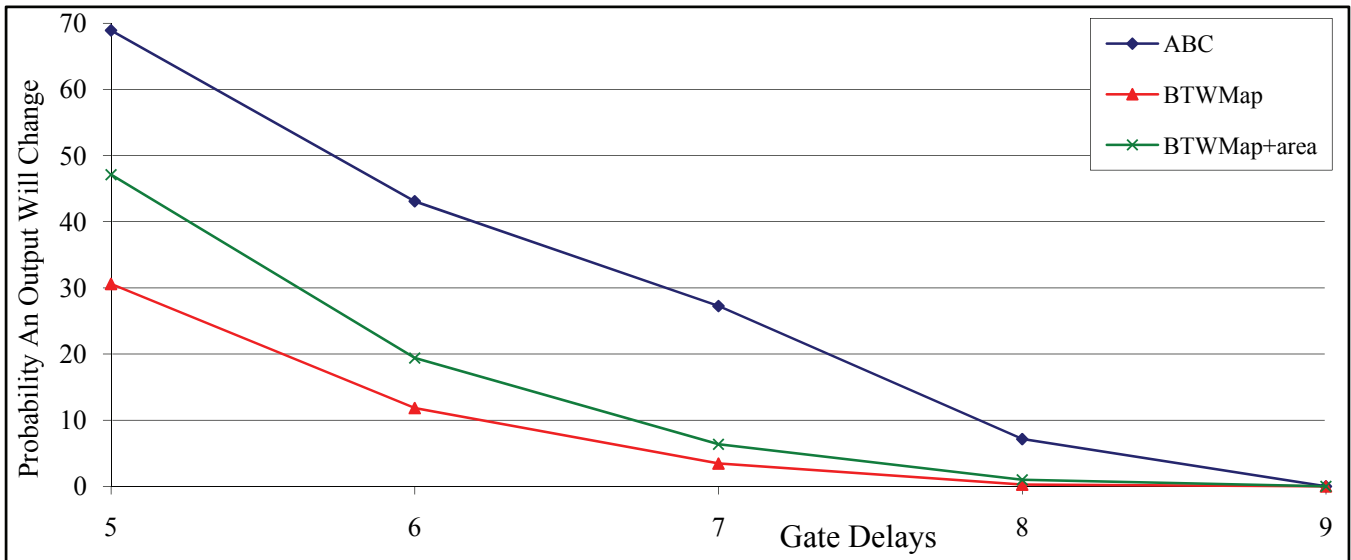


Figure 8. A switching activity comparison on the MCNC benchmark PDC.

4.3. Performance Improvements

The results in Table 3 examine the performance difference between circuits optimized by Berkeley’s ABC mapper and our BTWMap after they are implemented using the stallable FSM architecture. To calculate the expected delay, formula (3) was used. These results show that these circuits have significant room for improvement in terms of the expected delay under the stallable FSM architecture — 13% improvement in expected delay by BTWMap compared to ABC and 11% by BTWMap+area (but with 15% area reduction as seen in the next subsection).

Table 3. Expected delay comparison.

Circuit	Expected Delay			Ratio	
	ABC	BTWMap	BTWMap+area	BTWMap	BTWMap+area
alu4	7.00	6.33	6.43	90.4%	91.9%
apex2	6.83	6.39	6.60	93.6%	96.7%
apex4	7.00	6.09	6.09	87.0%	87.0%
clma	8.44	8.18	7.98	96.9%	94.6%
misex3	6.00	6.00	6.00	100.0%	100.0%
pdc	8.45	6.53	7.16	77.3%	84.8%
s298	2.30	2.01	2.01	87.0%	87.0%
s38417	8.00	6.00	6.00	75.0%	75.0%
s38584.1	6.35	5.27	5.32	83.0%	83.8%
seq	6.00	4.86	5.25	80.9%	87.5%
spla	7.28	6.48	6.68	89.0%	91.7%
Geomean				87.0%	88.8%

4.4. Area Penalty

By only optimizing depth-weighted switching probabilities, BTWMap incurs a considerable penalty of a 26% increase in area, as seen in Table 4. The problem is that BTWMap does not consider area as its first optimization goal, so when it has the choice between using a larger LUT with higher FO or one with better switching, it always chooses the latter.

Table 4. Area comparison.

Circuit	Area Increase			Improvement	
	ABC	BTWMap	BTWMap+area	BTWMap	BTWMap+area
alu4	722	922	776	27.7%	7.5%
apex2	972	1200	1040	23.5%	7.0%
apex4	793	1028	842	29.6%	6.2%
clma	4216	5674	5091	34.6%	20.8%
misex3	742	899	819	21.2%	10.4%
pdc	2234	2934	2498	31.3%	11.8%
s298	44	50	49	13.6%	11.4%
s38417	2909	3972	3234	36.5%	11.2%
s38584.1	3592	4543	3856	26.5%	7.3%
seq	1000	1258	1111	25.8%	11.1%
spla	2127	2735	2399	28.6%	12.8%
Geomean				26.4%	10.1%

From Table 4, one can see that BTWMap+area is effective in area and performance trade-off. If area is ignored and switching probabilities are optimized as much as possible, the result is a 26% increase in area. On the other hand, using the information of the given target clock period and some clever area/performance tradeoffs, the algorithm can reduce the area penalty by 15%, resulting in a 10% area overhead compared to ABC, yet still produces more than an 11% reduction of the expected delays. We also conducted an experiment to evaluate area-overhead by allowing BTWMap+area to completely ignore the switching activity. In this case, it would still have a 3% increase in area, and we believe it is due to how the circuit is traversed.

The BTWmap algorithm does come at an inherent runtime cost (due to the simulation) compared to that of ABC. But BTWMap is still able to complete each design in less than five minutes (on a 3.2 GHz Xeon Linux machine).

5. FUTURE WORK

We plan on performing the following tasks to further the study of better than worst-case synthesis:

1. Implement several designs under the stallable FSM architecture on an FPGA with two different clocks (one for the main registers and the other for the shadow registers). This will help evaluate the actual performance gains.
2. Go beyond the limits of the stallable FSM architecture by duplicating some of the logic. This will allow the circuit to go beyond the maximum clock imposed by the design of the stallable FSM architecture.
3. Investigate how other steps in the design flow can help to optimize circuits for the expected delay, from high-level synthesis all the way down to placement and routing.
4. Further reduce area overhead while improving the expected delay of BTWMap.

6. CONCLUSION

In this paper we have presented a novel methodology for measuring performance during logic synthesis for better than worst-case architectures, including the Razor architecture and its generalized version -- the stallable FSM architecture. We discussed our better than worst-case technology mapper BTWMap and its area recovery mechanism. We also presented how well the algorithm performed, using our performance measurement, on the MCNC benchmark suite. Using our algorithms, BTWMap and BTWMap+area, we were able to reduce the expected delay by 13% and 11%, respectively, on the set of circuits suitable for the stallable FSM architecture implementation. BTWMap+area is able to reduce the area overhead from 26% to 10% with minimal (less than 2%) increase in the expected delay.

Both BTWmap and BTWMap+area are available for download at: http://cadlab.cs.ucla.edu/software_release/btwmap/

7. ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under CCF-0530261.

8. REFERENCES

- [1] T. Austin, V. Bertacco, D. Blaauw and T. Mudge, "Opportunities and Challenges for Better Than Worst-Case Design," *Asia-South Pacific Design Automation Conference*, Jan. 2005.
- [2] T. Austin and V. Bertacco, "Deployment of Better Than Worst-Case Design: Solutions and Needs," *International Conference on Computer Design*, pp. 550 - 558, 2005.
- [3] M. Berkelaar and J. Jess, "Gate sizing in MOS digital circuits with linear programming," *Design Automation Conference*, pp. 217-221, Mar. 1990.
- [4] S. Bhanja and N. Ranganathan, "Switching Activity Estimation of VLSI Circuits Using Bayesian Networks," *IEEE Transactions on VLSI Systems*, vol. 11, no. 4, pp. 558-567, Aug. 2003.
- [5] S. P. Boyd, S. Kim, D. D. Patil, and M. A. Horowitz, "Digital Circuit Optimization via Geometric Programming," *Operations Research*, vol. 53, no. 6, pp. 899-932, Nov. 2005.
- [6] D. Chen and J. Cong, "DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs," *IEEE Transactions on Computer-Aided Design*, pp. 752-759, 2004.
- [7] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *IEEE Trans. Computer-Aided Design*, pp. 1-12, 1994.
- [8] J. Cong, J. Peck, and Y. Ding, "RASP: A General Logic Synthesis System for SRAM-Based FPGAs," *International Symposium on Field-Programmable Gate Arrays*, pp. 137-143, 1996.
- [9] J. Cong, C. Wu, and E. Ding, "Cut Ranking and Pruning: Enabling A General And Efficient FPGA Mapping Solution," *International Symposium on Field-Programmable Gate Arrays*, pp. 29-35, Feb. 1999.
- [10] G. F. Cooper, "The computational complexity of probabilistic inference using Bayesian belief networks," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 393-405, Mar. 1990.
- [11] K. De and P. Banerjee, "PREST: A System for Logic Partitioning and Resynthesis for Testability," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 4, pp. 514-525, Dec. 1993.
- [12] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," *36th Annual International Symposium on Microarchitecture (MICRO-36)*, Dec. 2003.
- [13] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System*, vol. 25, no. 11, pp. 2331-2340, Nov. 2006.
- [14] A. Saldanha, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Multi-level logic simplification using don't cares and filters," *Design Automation Conference*, pp. 277-282, Jun. 1989.
- [15] W. Shi and Z. Li, "A fast algorithm for optimal buffer insertion," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 879-891, June 2005.
- [16] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," <http://www.eecs.berkeley.edu/~alanmi/abc/>