Logic Synthesis for Better Than Worst-case Designs

Jason Cong[1,2] and Kirill Minkovich[1]
[1]Computer Science Department
University of California, Los Angeles
[2]California NanoSystems Institute
Los Angeles, CA 90095, USA
{cong, cory_m}@cs.ucla.edu

ABSTRACT

In this paper we present a novel metric for measuring and optimizing the performance of circuits that operate with the clock period smaller than the worst-case delay. In particular, we developed an efficient logic optimization operation "balance" and a library mapping algorithm named BTWLibMap. Together they are able to reduce the probability of a timing error by 2.3X while only incurring a 4% area overhead.

INTRODUCTION

As transistor sizes scale down to the nanometer range, many new design challenges occur, and delay variation is one that may significantly limit circuit performance. Delay variation is mainly caused by increasing the environmental and process variation, as well as data 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 further shrink, the process variation problem becomes greatly amplified. In the conventional design style which only considers 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 manufacturers speed-bin their products by selling different speed grades of the same chip. But most ASIC vendors do not have the luxury of selling different versions of the same product, and in turn suffer significant losses in potential circuit performance caused by their conservative models.

There is a common consensus that process variation is going to get much worse as we approach the limit of CMOS scaling and begin exploring different nanotechnologies as alternatives to CMOS scaling. We believe that a fundamental approach to dealing with design uncertainty is architectural and logic design innovations that will allow us to design and optimize circuits that can operate with better (smaller) than worst-case delay. In fact, the better than worst-case delay methodology may improve circuit performance *even in the absence of process variation*, as circuit delay may vary depending on the values of its inputs, which we call *data variation*.

Recently, a whole set of architectures have been developed that operate with better than the worst-case delay. These range anywhere from asynchronous designs [6] to Razor [1] and its counterparts (TED [12], SPRIT^{3}E [11], etc.). These architectures might vary greatly, but they have one thing in common   than theependence on typical-case delay. To familiarize the reader to architectures dependent on typical-case delay, we first discuss the Razor architecture [1], which provides a good motivation for us to better understand the optimization opportunities of clocking the circuit at a frequency higher than that determined by the worst-case delay.

The Razor architecture was originally developed for microprocessors with a 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 latches data which is not stabilized during the given clock period, signals an error, and stalls the pipeline. The exact error handling pipeline stage (originally presented in [1]) is shown in Figure 1. The shadow latch is clocked with a delayed clock, which allows it to catch any errors caused by the main flip-flop being clocked earlier than the worst-case delay. 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.

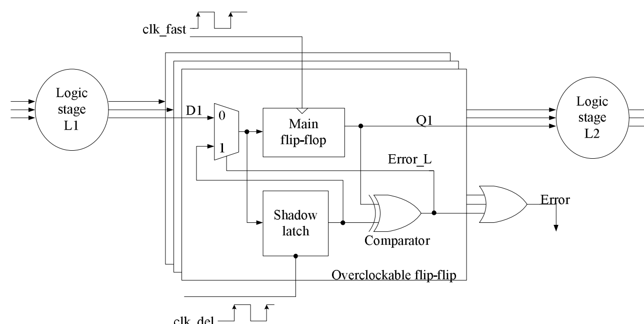

Figure 1. Razor stage with shadow latches and control lines.

Although good progress has been made on architectures for *BTW (Better Than Worst-case)* designs, few studies are available for the logic optimization for BTW. Standard logic synthesis operations aim at generating a circuit with the minimum worst-case delay, while optimizing area usually leads to a good delay in a placed and routed solution. However, such optimizations may not be as useful for BTW architectures, since their operating frequencies are not determined by the worst-case path delay, but are instead more closely correlated to the typical-case delay. The only study known to us is the one we presented in [4] where only the FPGA map algorithm was modified with a much higher area overhead and only applied to favorable circuits.

In the remainder of this paper we will first present a method for measuring the quality of solution for circuits optimized for better than worst-case delay. We will then present two methods (a logic optimization and a library mapper) for improving the performance of circuits implemented using BTW architectures. All the methods are evaluated on the 20 largest MCNC benchmark circuits.

DEFINITIONS AND PROBLEM FORMULATION

To understand the complex notion of measuring delay on a timing error-resistant architecture, one first has to understand a simpler notion of propagation delay. For a particular input sequence, the *propagation delay* is the maximum time for any gate output to transition to its final value under the given input sequence. Given the previous input $i_p=[i_1,i_2,\ldots,i_n]$ and the current input $i_c=[i'_1,i'_2,\ldots,i'_n]$, 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 2, going from input [1,1,1,1] to [1,1,1,0] takes 8 ns, while the transition from [1,1,1,0] to [1,1,0,0] takes 0 ns, since there is no signal transition at the output f.

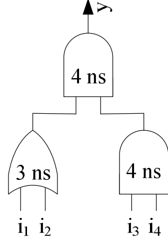

Figure 2. An example showing transition delay differences.

For conventional architectures the performance of the circuit is only dependant on the worst-case delay, or the static longest path in the logic network, which can be easily calculated in linear time using a simple topological traversal. For the BTW architectures, on the other hand, the performance is not only dependent on the worst-case delay but also on the error rate. Specifically, the Razor architecture [1] only uses extra clock cycles when it sees a timing error. Thus, the performance is determined by the likelihood of such an error. For this reason, we use the *error probability*, defined below, as a metric for measuring the performance of a BTW circuit under a given input sequence.

Error Probability

The *error probability* determines the performance of circuit C that will be implemented in an error recovery scheme. Referring to Figure 1, let P_{wrong}^{clk} be the probability of circuit switching at time $d = 1/clk$ or later. Now, if we estimate the fast clock period (*clk_fast*) on a BTW architecture, then the error probability (the probability of a timing error) is just $P_{wrong}^{clk_fast}$.

$$P_{wrong}^{clk_fast} = Pr\big(C(x)_{clk_fast} \; = \; C(x)_{clk} \;\; | \; x \in \text{inputs}\big) \qquad (1)$$

To calculate $P_{wrong}^{clk_fast}$ we need to know the switching probability of each gate. But 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 Boolean one at the last time interval is non-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, a partial simulation is used for run time and accuracy tradeoff. That is, we use the simulation-based solution to compute the switching probability under a given sample input sequence of significant length. Since each gate has discrete switching times, transitions can only happen at discrete times in the clock period. We keep track of the switching probabilities at all the possible switching times by establishing a vector for each node called the *switching probability vector* (as seen in equation (2) for node n and at time d). We can then sum this vector from time 0 to the maximum delay of the node n, d_m (shorthand notation for $d_m(n)$).

$$switching_probability(n, d) = \frac{number_transitions(n, d)}{\#\ vectors\ simulated} \qquad (2)$$

This metric can also be used to measure intermediate circuit representations and to measure performance of BTW circuits optimized for power or for latency (both of which are dependent on the error rate). But to make an accurate comparison we need to choose an accurate *clk_fast*. This is done by first examining the original circuit and choosing the smallest *clk_fast* for which the error is less than 5%. In some instances, any increase would lead to errors beyond 5% and in these instances we chose the minimal *clk_fast* for evaluation purposes. We then used this delay to extract $P_{wrong}^{clk_fast}$ of two synthesis solutions and used that number for our comparison.

BALANCE DECOMPOSITION FOR BTW

Given a specific metric, circuit optimization can be broken down into two classes. The first class of methods converts a circuit into an alternative representation (like AIG (AND-INV Graph) construction or library mapping) while optimizing some metric. The second class transforms the circuit into another one in the same representation, and this is the class we will cover in this section. The transformation operations in this class can be further broken down into two groups: algorithms that decompose nodes and algorithms that combine nodes (cleanup algorithms were purposely ignored). Combining algorithms are mainly used for creating a better environment for decomposition algorithms. For this reason, we will focus on adapting decomposition algorithms for the BTW architecture.

Most of the decomposition algorithms work in a similar fashion. For each node/function they encounter, they generate a set of possible decompositions, assign costs to each one, and pick the best one. Inevitably, there will be instances where the algorithm can choose between multiple decompositions of equal cost. This creates an opportunity to break these ties and hopefully reduce the error probability. To explore the solution space we choose to evaluate three possible ways to modify *balance* to improve the quality of solution.

The version of *balance*, that we used as the basis, was presented in [8] as a delay minimizing decomposition method. It was originally presented in a number of other logic synthesis systems with different names, such as [10] and [3]. This version assumes that the input circuit is an AIG and creates an equivalent AIG with minimum gate delay. Balancing is performed in topological order (PI to PO), where for each AND-gate the minimum delay tree-decomposition is selected. The runtime performance and results of this simple operation are so good that it is incorporated into every mapping algorithm in ABC [7] (for both FPGA and ASIC library mapping).

When looking at the internals of the balancing algorithm, we noticed that for the average circuit, 10% of the comparisons resulted in a tie. To explore the solution space, three different methods were created to improve balance. The first and simplest method was to pre-compute the original cost function from [4] (equation (3) listed below) for each node and use it as the tie breaker. This resulted in a 23% reduction in the probability of a timing error. For the second attempt we performed simultaneous simulation and balancing, which maintained an accurate cost function and resulted in a 31% reduction. At this point *balance* still has a lot of ties, but it also had simulation data for each node. To take advantage of this, the expected value for each node (using equation (4) listed below) was calculated and used as a second tie breaker. The *expected_value_cost* gives a smaller cost to nodes that have higher probability of being 1 or 0 and a larger cost to nodes that are more likely to change. Nodes with low cost are **not** likely to change, so clocking them early is not likely to cause an error. This final combined method resulted in a 38% (shown in TABLE I) reduction in the probability of a timing error without a noticeable change in area (in fact, area decreased by 0.1%). It is interesting to note that all three methods have a similar run time performance, and the final method is able to achieve a 61% improvement without any cost (in terms of area or depth).

We also studied enhancements of other logic optimization operations, such as fx (fast-extract) and refactor, for the BTW architecture. However, due to the page limit, we left them out of this paper.

$$cost(n) = \sum_{delay\ d=0}^{\max delay\ d_m} d^{\alpha} \cdot switching_probability(n, d) \qquad (3)$$

$$expected_value_cost(n) = .5 - |.5 - \Pr(n = 1)| \qquad (4)$$

TABLE I
Probability of Error Comparison for *Balance*

	Area			*Pr Error*		
Circuit	Orig.	New	Ratio	Orig.	New	Ratio
alu4	1,577	1,576	1.00	11.08	5.30	0.48
apex2	1,973	1,971	1.00	11.67	6.51	0.56
apex4	1,533	1,536	1.00	7.74	0.60	0.08
clma	10,127	10,127	1.00	3.67	3.70	1.01
misex3	1,459	1,457	1.00	3.54	1.45	0.41
pdc	4,974	4,990	1.00	20.98	4.63	0.22
s298	101	99	0.98	0.07	0.07	1.00
s38417	8,161	8,157	1.00	0.02	0.02	1.00
s38584.1	8,957	8,954	1.00	3.46	4.47	1.29
seq	2,091	2,089	1.00	2.95	0.77	0.26
spla	4,743	4,747	1.00	4.16	1.92	0.46
bigkey	2,862	2,863	1.00	10.93	11.67	1.07
des	3,017	3,016	1.00	64.02	31.12	0.49
diffeq	1,877	1,876	1.00	3.29	3.29	1.00
dsip	2,522	2,522	1.00	1.60	1.60	1.00
elliptic	748	748	1.00	0.50	0.50	1.00
ex1010	1,952	1,954	1.00	77.08	67.39	0.87
ex5p	990	993	1.00	9.92	5.92	0.60
frisc	7,830	7,815	1.00	4.48	4.40	0.98
tseng	2,386	2,387	1.00	0.09	0.09	1.00
Geomean			0.999			0.623

Library Mapping for BTW

Library mapping is the final step of logic synthesis; it maps an optimized Boolean network to a network of library gates. The goal of traditional delay-oriented library mapping is to minimize the total area of the gates needed to cover all the logical elements in the circuit while achieving the optimal mapping delay. The goal of BTWLibMap, for the better than worst-case mapping formulation described in this paper, is to minimize the probability of a timing error while keeping any area increase to a minimum and not increasing the worst-case delay.

We will use the ABC mapper as the foundation for our mapper because of its impressive area and delay performance [7] and the availability of the source code. Before any mapping is done, the mapper creates a set of supergates by combining multiple library gates until a fan-in limit is reached. In our implementation we used 5-input supergates and the standard cell library, *mcnc.genlib*, from the SIS distribution [9]. After the supergates are created, the circuit is decomposed into an AIG and all possible 5-input cuts are computed. The original library mapping procedure was divided into four mapping phases, a single delay-optimal phase followed by three area recovery phases. Our optimization was inserted as a fifth phase to allow a better comparison since no new cuts had to be computed. During this phase we performed simultaneous simulation and mapping.

Simulation

In order to optimize for BTW architectures, we first need to know the switching probability vector for each node. The switching probability vector represents the probability that a node will switch at a certain time. To maintain accuracy while being able to store the vector, we used the actual delays from the library with a resolution of .1 ns (which was the resolution of our library). Similar to the work in [4] for FPGAs, we found that a good tradeoff between accuracy and runtime would be to simulate 100,000 random inputs. [4] also had multiple iterations of simulating a small subset of values and using that information for mapping. The problem with that approach is that during each iteration the mapper accesses either a small subset of the data (only 256 of 100,000 values) from the current iteration, or stale data from previous iterations.

In this work we are more interested in accuracy, and thus chose to perform mapping and simulation in a single iteration. This allows us to manage the area increase while also maintaining a much more accurately transition function. However, due to memory constraints, our algorithm is not able to store all the data for all the nodes at the same time. Luckily, ABC maps the nodes in a topological order, PI to PO, which allowed our algorithm to free the simulation data from nodes that will not be used any more. By freeing up the unusable data, we are able to reduce the memory usage by 4x and all the benchmarks are easily executed within 2.8 GB of memory. Although it was not necessary in our case, the memory usage could be greatly reduced by either decreasing the resolution or by only keeping the relevant parts of the switching probability vector (relevancy will be defined in *Target Delay* section below).

Mapping

The mapping of a single node can be broken into two steps: finding the best supergate (a set of library gates) for a cut and finding the best cut (see [2] for definition) for a node. Since our library is complete, every non-trivial cut has a supergate implementation. In both of these two steps, our simulator is used to generate a switching probability vector to determine the best choice. In order to compare the vectors of two nodes, we created a cost function (equation (6) listed below) by enhancing equation (3) to consider the arrival time (i.e. the maximum delay), d_m, of the node. This enhancement allows our new cost function to prefer nodes that have lower arrival times. For example, if two nodes, *a* and *b,* had the same cost using equation (3) but node *a* had an earlier arrival time, then using our new cost function, node *a* would have the smaller cost.

Multi-Level Mapping

We noticed that optimizing circuits with unit delay is significantly easier since switching is limited to small number of time intervals, making each improvement very significant. However, with real delays used in standard libraries, the switching vectors grew 15X (as compared to [4]), thus making any reduction of switching in a single interval not very significant. If the switching vectors use unit delay, optimization would be simpler but the result wouldn't be very accurate. On the other hand, if switching vectors use real delay, optimization is much harder but the result would be accurate. To exploit both of these cases, BTWLibMap performs two iterations, first a coarse-grain mapping, followed by a fine-grain mapping. For the coarse-grain mapping, we round the delay values for a 3X reduction of switching vectors (.3ns rounding for our library). For the fine-grain mapping, no rounding is performed.

Area/Performance Tradeoff

If the goal of mapping was only to reduce the switching activity of each node, a huge area increase would be experienced. On the other hand, if no area increase was allowed, then the algorithm would become too restricted and the switching activity would barely improve. In order to reach an acceptable balance between area and switching activity, we used equation (5) to specify the amount of acceptable area increase for a specific cost decrease. This allowed the end user to adjust the acceptable amount of area increase.

$$\beta \cdot \frac{area_{new}}{area_{old}} < \frac{cost_{old}}{cost_{new}} \quad (5)$$

Target Delay

To further reduce area, we took the concept of target delay from [4] and applied it to library mapping. During mapping our algorithm considers both the target delay and the maximum mapping delay

(which can be computed easily in a cut enumeration framework [5]) to achieve the best results for the minimal area increase. For example, if the circuit has a maximum delay of 10 ns and a target delay of 7 ns is specified, the algorithm will try to ensure that the outputs do not change after 7 ns, while preserving the maximum mapping delay of 10 ns.

Given a target delay, g (1/*clk_fast* from Figure 1), BTWLibMap ignores all the switching that happens before g by calculating the switching cost starting from g (instead of starting from 0). This greatly increases the number of cuts where the mapper can use area to break the ties in the switching costs. In order to propagate the target delay to each node, we use the *switching threshold*. The switching threshold of a node n is a number *ST(n)* which allows the node to ignore all the switching before time *ST(n)*. The difference between the switching threshold of a node and the target delay of a circuit is analogous to the difference between the required time of a node and the maximum delay of a circuit. The switching threshold calculation and propagation is the same as the required time calculation and propagation used in [2].

Combining all these ideas together produces the new cost function (Equation (6)) that is used during mapping.

$$cost(n) = d_m \cdot \sum_{delay\ d=ST(n)}^{\max delay\ d_m} d^{\alpha} \cdot switching_probability(n, d) \quad (6)$$

TABLE II
Library mapping comparison

Circuit	Area			Pr Error		
	Orig.	New	Ratio	Orig.	New	Ratio
alu4	2,908	3,097	1.06	3.61	0.32	0.09
apex2	3,779	3,988	1.06	3.98	2.15	0.54
apex4	2,849	2,858	1.00	18.19	12.04	0.66
clma	18,463	18,896	1.02	4.78	3.03	0.63
misex3	2,891	2,950	1.02	3.29	1.09	0.33
pdc	9,796	10,298	1.05	4.77	0.51	0.11
s298	199	201	1.01	3.93	1.4	0.36
s38417	15,609	16,088	1.03	4.03	1.02	0.25
s38584.1	16,431	17,499	1.06	4.91	0.68	0.14
seq	4,090	4,212	1.03	3.84	2.09	0.54
spla	9,451	10,034	1.06	4.73	1.83	0.39
bigkey	5,849	5,879	1.01	7.28	7.33	1.01
des	6,061	6,516	1.08	37.92	19.22	0.51
diffeq	3,444	3,717	1.08	4.93	1.68	0.34
dsip	5,014	5,523	1.10	7.3	1.91	0.26
elliptic	1,777	1,882	1.06	4.58	4.9	1.07
ex1010	3,544	3,604	1.02	2.69	5.2	1.93
ex5p	2,146	2,102	0.98	8.38	4.35	0.52
frisc	12,669	13,351	1.05	4.95	3.41	0.69
tseng	4,409	4,477	1.02	4.99	4.65	0.93
Geomean			1.040			0.435

Mapping Results

In our mapping algorithm we can tune two variables, α and β, both of which allow for a tradeoff between quality of solution and area. By varying these variables, we can make the area increase range from 0% to 29% which would correspond to a 2% to 73% error reduction, respectively. Instead of using either extreme, we chose α and β to be 7.0 and 2.7, respectively, which corresponded to the largest error reduction without a significant area increase.

When we modify the ABC mapping procedure, *map*, which consists of balancing followed by mapping, by using the modified version *balance* and BTWLibMap, we are able to see a 56.5% reduction in error probability (a 2.3X improvement), a 0.1% increase in worst-case delay, and only a 4% area increase (as seen in TABLE II).

Summary and Conclusions

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 other BTW architectures. We discussed our modified versions of logic optimization operation, *balance,* and the better than worst-case library mapper BTWLibMap. We showed how to improve decomposition algorithms using a novel cost function. We discussed how we met the challenges of library mapping with improved cost functions, better memory management and multi-level mapping. Using BTWLibMap with *balance*, we were able to reduce the probability of a timing error by 56.5% with only a 4% area overhead.

The modified ABC package is available for download at:
http://cadlab.cs.ucla.edu/software_release/btw/

Acknowledgements

Financial support from NSF (US) award CCF-0530261, SRC contract 2006-TJ-1460, and support from NSFC (China) for international collaboration are greatly acknowledged.

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] 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.
[3] K. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar, “DAG-Map: Graph-based FPGA technology mapping for delay optimization,” *IEEE Design and Test of Computers*, vol. 9, no. 3, pp. 7-20, Sept. 1992.
[4] J. Cong and K. Minkovich, “Mapping for better than worst-case delays in LUT-based FPGA designs,” *International Symposium on Field-Programmable Gate Arrays,* pp. 56-64, Feb. 2008.
[5] 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.
[6] S. Hauck, “Asynchronous design methodologies: an overview,” *Proceedings of the IEEE*, vol. 83, no. 1, January 1995.
[7] A. Mishchenko, S. Chatterjee, R. Brayton, and M. Ciesielski, “An integrated technology mapping environment,” *Proceedings International Workshop on Logic & Synthesis,* pp. 383-390, Jun. 2005.
[8] J. Cortadella, “Timing-driven logic bi-decomposition,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675-685, Jun. 2003.
[9] E. Sentovich, et al. “SIS: A system for sequential circuit synthesis,” Tech. Rep. UCB/ERI, M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.
[10] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, “Timing optimization of combinational logic,” *International Conference on Computer-Aided Design*, pp.282-285, Nov. 1988.
[11] V. Subramanian, M. Bezdek, N. Avirneni, and A. Somani, “Superscalar processor performance enhancement through reliable dynamic clock frequency tuning," *Annual IEEE/IFIP International Conference on Dependable Systems and Networks,* pp. 196-205, Jun. 2007.
[12] Y. Su, P. Chang, S. Chang, and T. Hwang, “Synthesis of a novel timing-error detection architecture,” *ACM Transactions on Design Automation of Electronic Systems,* vol. 13, no. 1, pp. 1-14, Jan. 2008.