

# A Variation-Tolerant Scheduler for Better Than Worst-Case Behavioral Synthesis

Jason Cong                  Albert Liu                  Bin Liu  
Computer Science Department, UCLA  
{cong, albliu, bliu}@cs.ucla.edu

**Abstract** – There has been a recent shift in design paradigms, with many turning towards yield-driven approaches to synthesize and design systems. A major cause of this shift is the continual scaling of transistors, making process variation impossible to ignore. Better than worst-case (BTW) designs also exploit these variation effects, while also addressing performance limits due to worst-case analysis. In this paper we first present the variation-tolerant stallable-FSM architecture, which provides fault detection and recovery, allowing circuits to be clocked at better than worst-case delays. Then we propose the BTW scheduler, a 0-1 integer linear programming (ILP) scheduling algorithm with the objective of minimizing the expected latency, to provide a high-level synthesis aid for the stallable-FSM architecture. We implemented the algorithm and ran it through many benchmarks, comparing the results with scheduling algorithms based on worst-case analysis. Our results were promising, showing up to 41% latency reduction for the BTW scheduler, and up to 43% latency reduction when combined with the variation-tolerant architecture.

## 1. Introduction

Traditionally, deterministic worst-case analysis has been used to estimate circuit performance. However, as technology continued to scale and transistor sizes decreased, concerns regarding the effects of process variation have risen. In [2], the panelists addressed the idea of variation-aware analysis, all agreeing that such a shift is necessary and inevitable. Many have begun to use statistical static timing analysis (SSTA) to model the delay and power behavior of circuits, where the overall performance is calculated by propagating distributions, rather than a fixed delay. With the degree of variability increasing, worst-case analysis will lead to overly conservative estimations, resulting in excess allocation of resources. Figure 1 shows a Gaussian distribution of the delay of a multiplier from [4]. If we assume a 5ns clock period, the worst-case analysis will allocate 9 clock cycles for the operation to complete. Yet it is evident that only about 15% of the time will the operation actually take 9 clock cycles to complete, while 85% of the time the operation will complete within 8 clock cycles in a fault environment with data variations.

Variable delays of a combinational functional unit can be realized in asynchronous designs, as these designs naturally use a ‘done’ signal to indicate the completion of a computation. Special encodings can be used to represent both the value and its validity [18]. An alternative is to use current sensors in the circuit to detect the stability of signals [19]. Nowick et al. proposed speculative completion, in which predefined logic is used to determine the delay for a particular input [20]. All these styles require nontrivial additional logics or sensors.

In Synchronous designs, variation awareness is particularly relevant for high-level synthesis (HLS). HLS has been a well-studied problem [11], commonly focusing

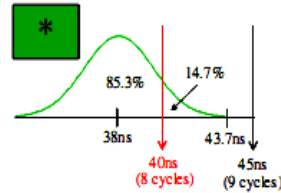


Figure 1. Delay distribution of a multiplier

on data flow designs from the behavior level and attempting to optimize performance or area through various scheduling, binding, and resource sharing techniques. It is especially beneficial to address variation from the behavior synthesis level, giving the designer more

opportunities for optimization and yield maximizing. ILP-based methods, although known for its inability to scale to larger problem sizes, have often been proposed to find the optimal solution for high-level synthesis. Chaudhuri et al. [15] gives a formal analysis of an ILP-based formulation, and presents ways to reduce the computation time and increase efficiency. In [11], ILP formulation was presented as a solution for resource constraint scheduling; [16] uses an ILP-based formulation to do scheduling and resource binding for power optimization. The rising concern about the effects of process variation has led to a number of variation-aware ILP-based high-level synthesis frameworks. An ILP formation that seeks to maximize reliability, given area constraints, is presented in [13]; and in [14] a similar ILP formulation is presented with the specific resource binding constraints. In addition, [12] proposed a yield-driven ILP formulation with the capability for both area constrained yield optimization, or yield constrained area minimization. Other non-ILP-based variation-aware high-level synthesis methods have also been proposed. Xie et al. [6] presented a simulated annealing (SA) based HLS framework, where clock selection was included as part of the synthesis flow. In [5], a SA based algorithm was also used, but both the timing and power yield were taken into consideration in the cost function of their synthesis flow. A heuristic-based approach was presented in [4], pointing out how operation binding can affect the total yield during synthesis.

However, with yield-driven HLS algorithms, the objective is set either to maximize or to satisfy the yield constraint. In either case, although limited, the correctness of the design could be compromised with the possibility of timing fault. Our proposed research focuses on extending the widely used synchronous design methodology to cope with a high degree of variation, guaranteeing that the design is free of timing faults at a given frequency. In particular, we plan to develop a new methodology and the associated synthesis tools to allow a synchronous design with its clock operating at better than the worst-case delay [17]. The stallable-FSM architecture, an architecture built on methodologies proposed in [10], allows us to do this

because it is able to detect and correct faults. Our objective is to minimize the expected latency, accounting for both variation and fault recovery, producing a latency-optimized variation-tolerant schedule. To the best of our knowledge, this is the first time that such a scheduling problem is formulated and studied. In addition, the variation-tolerant architecture can improve circuit performance even in the absence of process variation. As discussed in [7], different inputs into the same gate will cause different levels of switching, potentially resulting in varied propagation delays. Thus we have every reason to believe that the better than worst-case design approach will allow tremendous performance improvement in coping with all types of variation, including environmental, process, and propagation delay variations.

In this paper we specifically address the synthesis methodology that caters to the specific properties of our architecture. The rest of the paper is organized as follows: First we describe the variation-tolerant architecture in Section 2. Next, we formulate the better than worst-case (BTW) scheduling problem in Section 3, and propose an integer linear programming (ILP) based algorithm that determines the most suitable variation-tolerant schedule in Section 4. Our aim is to minimize the expected latency of the design accounting for penalties introduced from fault recovery. To determine the effectiveness of our proposal, the BTW synthesis tool was performed on a set of benchmarks, and the experimental results are discussed in Section 5. Finally, we give our conclusion and discuss further work in Section 6, and list out references in Section 7.

## 2. BTW Variation-Tolerant Architecture

The proposed approach to dealing with variation tolerance builds on the Razor architecture [10], which was originally developed to overcome both delay uncertainty in microprocessor designs with the well-defined pipeline architecture. Ernst et al. [10] presents a special Razor register that is able to detect faults in a circuit when the timing requirement of operations is not met. The idea is to have an extra set of registers clocked with the delayed clock that gives the operation extra time to compute the result. Using these registers, the Razor architecture can detect if one of the registers latches data which is not stabilized during the given clock period, signaling an error, and stalling the rest of the operations while propagating the correct value from the extra set of registers. Razor architecture was originally proposed for low power designs, based on the idea that as one may dynamically lower the voltage to a circuit, the Razor registers maybe be used to tolerate timing faults. It can be observed, however, that the same principles of this design can be applied more generally to synchronous circuits when dealing with an aggressively clocked circuit, allowing it to operate at a faster frequency, yet recovering faults when detected with some penalties.

Using this idea, a “*stallable finite state machine (FSM) architecture*” was proposed in [7], applying the ideas of Razor to a general synchronous circuit controlled by a FSM. With the stallable-FSM architecture, one may detect faults that occur in the system, stall the FSM, and then recover the

faults detected. With every fault detected and corrected, there will be 1 stall cycle penalty for the recovery cycle. After the stall cycle the FSM will resume and continue with the correct data value. Figure 2 shows a diagram of the stallable FSM architecture.

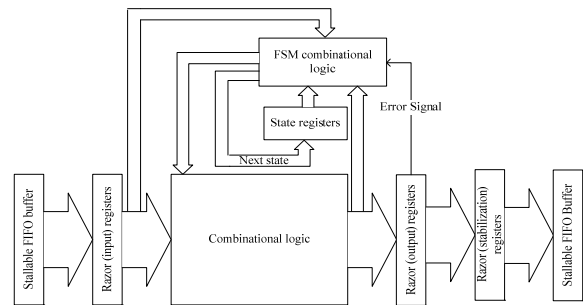


Figure 2. Stallable-FSM architecture

It is also worthy to note that the architecture will fit in well with the Globally Asynchronous Locally Synchronous (GALS) architecture [8] for Network-on-Chip (NoC). In a GALS architecture, two synchronous design blocks communicate via a two-stage asynchronous FIFO, allowing each block to operate on its own clock and pace. Thus, we can perform BTW synthesis on a local block, allowing it to operate at better than worst-case clock locally, and not affect the rest of the design.

## 3. Preliminaries and Problem Formulation

In this section we will explain the problem definition and the proposed ILP-based formulation for constraint scheduling that takes into account fault correction and delay variation.

### A. Motivational example

There have been several variation-aware High-Level Synthesis (HLS) algorithms proposed that seek to maximize the timing yield of a circuit caused by process variation [4, 14]. However, our stallable-FSM architecture is free from timing fault through fault recovery. As a result, we need new scheduling algorithms that cater to the specific properties of the stallable-FSM. For example, given the data flow graph (DFG) shown in Figure 3, Figure 3a shows a scheduling solution optimized for the worst-case delay (assuming the multiplier takes two clock cycles in the worst case). Applying better than worst-case design, we may schedule the multiplication in a single clock cycle. A latency-driven scheduling algorithm with timing yield optimization (as the one in [14]) will lead to a new scheduled DFG shown Figure 3b. However, we see that with error correction, that schedule could potentially need up to 7 clock cycles, including recovery if the multipliers in clock cycles 1, 2, and 3 all fail (e.g, requiring 2 clock cycles). The schedule in Figure 3c, which also uses 4 clock cycles in the best-case scenario, will only use up to a maximum of 6 clock cycles if all multipliers introduce

errors and need to be recovered. Hence we see the need for a new objective in HLS for our fault-free architecture.

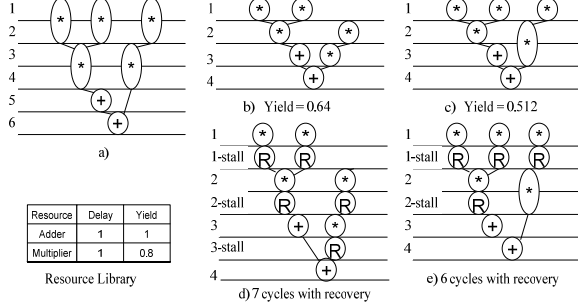


Figure 3a. Worst-case delay schedule; 3b. Yield-Driven schedule; 3c. BTW schedule; 3d. Yield-driven schedule with fault; 3e. BTW schedule with fault

## B. Preliminaries

With a given frequency, we define *timing yield* as the probability that a function unit will finish in the given amount of clock cycles. The resource library contains information regarding the delay distribution of function units, in which we can easily determine the timing yield with the given frequency and targeted clock cycles (e.g, in Figure 1, the timing yield is .853 when we allocate 8 clock cycles for the multiplier at 200MHz frequency). Next, we introduce the idea of *expected latency*, which is the expected clock cycles needed to carry out an iteration of the intended circuit. Essentially, the goal is to minimize the latency while factoring in possible stall cycles based the timing yield.

To measure the expected latency of an operation  $o$  given a frequency and the targeted clock cycles  $T$ , we use the following equation:

$$E_{f,T}(o) = T * p_{f,T}(o) + (T + 1) * \overline{p_{f,T}(o)} \quad (1)$$

where  $p_{f,T}(o)$  is the timing yield of the operation. When operation  $o$  does not complete with the allocated delay, then an extra stall cycle is allocated for the operation to complete. For example, if operation A is targeted to finish in 1 clock cycle and has a timing yield of 0.7, then the expected latency of the operation will be  $(0.7 * 1) + (0.3 * 2) = 1.3$ , since 70% of the time the operation will complete in 1 clock cycle, and 30% of the time we will need to use the extra stall cycle to recover a fault, needing 2 clock cycles total. Currently, due to the nature of our architecture, we can only recover timing faults up to a difference of 1 clock cycle from the targeted clock cycle.

To find the expected latency of a circuit, we calculate the expected latency of each clock cycle state, and then sum them all up. For each clock cycle state  $t$ , the yield is now the probability of all operations in this cycle completing on-time, and the targeted number of clock cycles is always 1 (we now use  $p(t)$  and  $E(t)$  instead of  $p_{f,t}(t)$  and  $E_{f,t}(t)$  since our formulation assumes a fixed frequency and targeted clock cycle is given). For an example, suppose a scheduled set of operations and resource library at a certain frequency

is given in Figure 4. For clock cycle state 1, the expected value  $E(1)$  is calculated as follows:

$$p(1) = p(A) * p(B) = 0.63$$

$$\overline{p(1)} = 1 - p(1) = 0.37$$

$$E(1) = 1 * p(1) + 2 * \overline{p(1)} = 1.37$$

The same calculations can be applied for clock cycle states 2 and 3 to find  $E(2) = 1.2$ ,  $E(3) = 1$ . So the total expected latency of this circuit will be  $E(1) + E(2) + E(3) = 3.57$ . This can be interpreted as the circuit taking approximately 7 clock cycles to complete 2 iterations, introducing 1 error (based on the yield) that will be corrected.

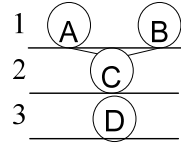


Figure 4a. Sample DFG  
4b. Sample resource library

Node	Delay	Yield
A	1	0.7
B	1	0.9
C	1	0.8
D	1	1

## C. Problem Formulation

With the variation-tolerant architecture described in Section 2 and the calculation of the expected latency described in Section 3b, we can formulate the **better than worst-case (BTW) scheduling problem** as follows:

**Given:** (1) a data flow graph (DFG)  $G(V, E)$  which represents the operations and dependencies in the circuit; (2) a set of scheduling constraints  $C$ , which includes dependency constraints, resource constraints, latency constraints and assignment constraints; (3) a resource library  $R$  containing the targeted clock cycle and yield of function units (can be calculated based on frequency and delay distribution).

**Goal:** Construct the state transition graph (STG) so that every operation is assigned to at least one state in the STG, all constraints in  $C$  are satisfied, and the expected latency of the circuit under the stallable FSM architecture is minimized.

## 4. Variation-Tolerant ILP Formulation

We are now ready to give the ILP formulation of our BTW scheduler. Table 1 lists the constants and the variables used in our formulation.

### A. Variable Definition

For our ILP formulation, we define two types of binary variables  $x$  and  $e$ , and one integer variable  $T$ :

- $x(o, t)$  : the last cycle of an operation  $o$  is scheduled at clock cycle  $t$ .
- $e(o)$  : operation  $o$  will not be given slack to guarantee its correctness.
- $T$ : the final latency (in clock cycles) of our scheduled circuit.

Decision variable  $x(o, t)$  was selected to be the last cycle of an operation because the stall recovery cycle always happens on the cycle after the last targeted cycle. So even if you have a multi-cycle operation, a fault will only be introduced after the last targeted cycle, so all the previous cycles have 100% probability of finishing on time in their respective scheduled cycles (e.g. in figure 3b, if the multiplier were a 2-cycle operation with 0.9 yield, the first clock cycle of the 2-cycle multiplier has a yield of 1, while the second cycle will have the 0.9 yield). Variable  $e(o)$  is the decision variable to decide whether there will be an opportunity for operation  $o$  to use extra slack as its recovery stall cycle. If  $e(o) = 1$ , that means no slack is given and the operation  $o$  is expected to complete in the given amount of cycles. If  $e(o) = 0$ , the operation will utilize time slack in the schedule and allocate an extra stall cycle for the operation to complete, making the yield of the operation 1 and increasing the delay of the operation by 1 extra cycle. If the timing yield of an operation  $o$  was originally 1, then  $e(o)$  will automatically be set to 1. The integer variable  $T$  is the minimized latency in our final scheduled circuit when no timing fault occurs.

## B. Constraint Generation

The first constraint that needs to be accounted for is the assignment constraint, which is expressed in the Equation (2)

$$\forall o : \sum_{t=1}^L x(o, t) = 1 \quad o \in [1, N_{ops}] \quad (2)$$

This means that each operation can only be scheduled in one clock cycle state. Next, Equation (3) is added to enforce the latency constraint, assuring that operations will not be scheduled over the final latency as we will minimize  $T$ .

$$\forall o : (1 - e(o)) + \sum_{t=1}^L t * x(o, t) \leq T \quad o \in [1, N_{ops}] \quad (3)$$

$(1 - e(o))$  signifies the fact that if  $e(o)$  is 1, then no time slack will be given, so we don't need to account for an extra cycle. However, if  $e(o)$  is 0, then we will need to account for the extra slack cycle given. Precedence constraints are enforced through Equation (4). The equation states that the number of clock cycles between a pair of dependent operations ( $Src, Des$ ) has to be greater than the delay of the destination operation plus the potential that the source operation might be given time slack.

Notion	Type	Definition
$t$	Index	Clock cycle stage
$o$	Index	Operation number
$r$	Index	Type of resource
$R$	Library	Resource library
$L$	Constant	Estimated latency
$N_{ops}$	Constant	Number of operations
$N_r$	Constant	Number of types of resources
$p(o)$	Constant	Yield of operation $o$ from library
$D(o)$	Constant	Latency of operation $o$ from library
$T$	Integer Variable	Total minimized latency
$x(o, t)$	Binary Variable	Assignment variable signifying that operation $x$ will complete in cycle $t$
$e(o)$	Binary Variable	Whether operation $o$ is allocated slack time. (1 = no, 0 = yes)
$Q(o, t)$	Binary Variable	$Q(o, t) = x(o, t) \& e(o)$
$Q(o, o', t)$	Binary Variable	$Q(o, o', t) = Q(o, t) \& Q(o', t)$

**Table 1 Variables used in ILP formulation**

$$\forall (o_{src}, o_{des}) \in G(V, E) : \quad (4)$$

$$\sum_{t=1}^L t * x(o_{des}, t) - \sum_{t=1}^L t * x(o_{src}, t) \geq D(o_{des}) + (1 - e(o_{src}))$$

Finally, it is also possible to add resource constraints into our formulation. In order to ensure correctness with resource sharing, the worst-case delay for each resource with a yield has to be used as the parameter instead of the targeted delay (see Section 5A). The method of resource binding and sharing is beyond the scope of this paper, but a resource constraint is modeled in Equation (5) for resources with yield, and Equation (6) for resources that are guaranteed to complete.  $D(r)$  is the targeted delay of the resource.

$$\forall r, t : \sum_{j=t-1}^{t+D(r)-1} \sum_{o=1}^{N_{ops}} x(o, j) \leq N_r \quad L \in [1, t]; r \in R_{yield} \quad (5)$$

$$\forall r, t : \sum_{j=t}^{t+D(r)-1} \sum_{o=1}^{N_{ops}} x(o, j) \leq N_r \quad L \in [1, t]; r \in R_{WC} \quad (6)$$

## C. Objective Function

The objective of the BTW scheduler is to minimize the expected latency of our schedule. As defined in Section 3b, expected latency is the sum of the expected latency in all clock cycle states. Each clock cycle state's expected latency  $E(t)$  was stated in Equation (1), but here we simplify it in Equation (7):

$$\begin{aligned} E(t) &= (1 * p(t)) + (2 * \overline{p(t)}) \\ &= p(t) + 2(1 - p(t)) = 2 - p(t) \end{aligned} \quad (7)$$

Now we can formulate our objective function as follows:

$$\min \sum_{t=1}^T E(t) = 2T - \sum_{t=1}^T p(t) \quad (8)$$

As defined in Section 3b,  $p(t)$  is the probability that all operations in clock cycle state  $t$  complete without fault. To find  $p(t)$ , we use Equation (9) :

$$p(t) = \prod_{o=1}^{N_{ops}} (1 - x(o, t) * e(o) * \overline{p(o)}) \quad (9)$$

This equation states that for all operations, if the operation is cycled at time  $t$  and given no slack for the stall cycle, then multiply the current expression by  $p(o)$ . If the operation is not scheduled at this cycle, or is given slack so the yield is now 1, or the operation itself has yield 1, then just multiply the current expression by 1, bypassing this operation (anything multiplied by 1 equals itself).

However, since integer linear programming is used, we need to linearize  $p(t)$ . First, because  $x(o, t)$  and  $e(o)$  are both binary variables, we can linearize the expression  $x(o, t) * e(o)$  with the technique presented in [12]. This allows us to replace  $x(o, t) * e(o)$  with  $Q(o, t)$  as stated in Equation (10).

$$\begin{aligned} \forall o, t: \quad & x(o, t) + e(o) - 1 \leq Q(o, t) \\ & Q(o, t) \leq x(o, t) \quad ; \quad Q(o, t) \leq e(o) \end{aligned} \quad (10)$$

In addition, Equation (9) can be multiplied out into Equation (11).

$$\begin{aligned} p(t) = & 1 - \sum_{o=1}^{N_{ops}} Q(o, t) * \overline{p(o)} + \\ & \sum_{o1=1}^{N_{ops}} \sum_{o2=o1+1}^{N_{ops}} Q(o1, t) * Q(o2, t) * \overline{p(o1)p(o2)} - \\ & \sum_{o1=1}^{N_{ops}} \sum_{o2=o1+1}^{N_{ops}} \sum_{o3=o2+1}^{N_{ops}} Q(o1, t) * Q(o2, t) * Q(o3, t) * \overline{p(o1)p(o2)p(o3)} + \dots \end{aligned} \quad (11)$$

Note that the coefficient of the terms is the probability that all the operations involve failing, and this will gradually get smaller due to more operations becoming involved. We see that for the coefficients of third order-terms, if the probability of each operation failing is 0.1, the coefficient would already be around 0.001, which is pretty insignificant. Therefore, we choose to ignore the terms from the third order and beyond, looking at only the first-order term and second-order term. Our objective function now becomes Equation (12) (the summation of clock cycle states can go up to  $L$  instead of  $T$  because all  $Q(o, t)$  in between time  $[T, L]$  will be zero):

$$\begin{aligned} \min 2T - T + \sum_{t=1}^L \sum_{o=1}^{N_{ops}} Q(o, t) * \overline{p(o)} + \\ \sum_{t=1}^L \sum_{o1=1}^{N_{ops}} \sum_{o2=o1+1}^{N_{ops}} Q(o1, t) * Q(o2, t) * \overline{p(o1)p(o2)} \end{aligned} \quad (12)$$

In addition, the first-order terms in our objective function can be simplified due to Equation (2). This is shown in Equation (13):

$$\begin{aligned} \sum_{o=1}^{N_{ops}} \sum_{t=1}^L Q(o, t) * \overline{p(o)} &= \sum_{o=1}^{N_{ops}} \sum_{t=1}^L x(o, t) * e(o) * \overline{p(o)} \\ &= \sum_{o=1}^{N_{ops}} e(o) * \overline{p(o)} \sum_{t=1}^L x(o, t) = \sum_{o=1}^{N_{ops}} e(o) * \overline{p(o)} \end{aligned} \quad (13)$$

So combining Equations (12) and (13), our objective function is finalized to be:

$$\begin{aligned} \min T + \sum_{o=1}^{N_{ops}} e(o) * \overline{p(o)} - \\ \sum_{t=1}^L \sum_{o1=1}^{N_{ops}} \sum_{o2=o1+1}^{N_{ops}} Q(o1, o2, t) * \overline{p(o1)p(o2)} \end{aligned} \quad (14)$$

Notice we use the same method as Equation (10) to linearize  $Q(o1, t) * Q(o2, t)$ . Intuitively, the first term is the final latency we seek to minimize. The second term expresses that the more slack we can find for operations, the better the object function becomes. The third term indicates that when you have 2 operations scheduled on the same cycle, you will have a smaller result, thus rewarding the decision to schedule operations in parallel.

## 5. Experimental Results

In this section we present the experimental results of our ILP scheduling. We first show the results of our BTW scheduler on a set of high-level synthesis benchmarks. Then we validate our variation-tolerant architecture by implementing the synthesis of the DES design onto Altera's Development and Education Board 2, which has a Cyclone II (2C35) FPGA [1].

### A. BTW Scheduler Results

To implement our BTW scheduler, we leveraged the xPilot behavioral synthesis framework [3]. We ran the BTW scheduling algorithm using a PC with an Intel Xeon 2.40GHz processor on a set of high-level synthesis benchmarks: differential equation solver (DES); 16-point elliptic wave filter (EWF); FFT filter; FIR16 filter; and autoregressive lattice filter (AR). We used the resource library from [14] to run our experiments, with each function

unit's delay distribution characterized in SPICE with the Monte Carlo method to model both intra-die and inter-die variations. Due to their formulation's dependence on resource binding to maximize yield, [14] uses a set of adders and multipliers to perform synthesis. Our formulation is independent of resource binding, and thus needs only one type of resource for each type of operation. We chose to use adder 1 and multiplier 1 in our experiment (shown in Table 2); however, any delay distribution can be used in our algorithm.

In Table 3 we show the results of our scheduler with resource constraints set at 4 multipliers and 4 adders. We compared our results to schedules using the worst-case deterministic delay (WC) performed by giving our ILP scheduler resources with the worst-case delays (Table 2). Due to concerns with the scalability of problem sizes for ILP, we also included the run time of each experiment (in seconds) in the last column. BTW is the minimum latency scheduled for the DFG to complete if there are no faults; BTW-exp is the expected latency for the DFG when factoring the possibility of encountering stall cycles for fault recovery. When comparing the expected latency versus the worst-case latency, up to 30.65% latency reduction, with an average around 22% reduction, was achieved using our scheduler.

Currently, a limitation of our architecture is in resource sharing. When resource constraint is imposed and a function unit is shared by multiple operations, the worst-case time has to be allocated for each function unit to complete an operation before the next scheduled operation can use that function unit. This is because our fault detection and recovery hinges on the extra cycle to check for faults. If the next operation changes the inputs to the function unit, there would not be enough time to perform fault detection. Hence, when compared to yield-driven

Resource	Latency	Yield
Adder (BTW)	1 (cc)	0.9
Multiplier (BTW)	4 (cc)	0.92
Adder (worst-case)	2 (cc)	1
Multiplier (worst-case)	5 (cc)	1

Table 2. BTW resource library

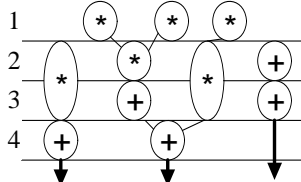


Figure 6. DES board design

schedulers such as [14] where correctness is not guaranteed, we are not able to achieve as high of a reduction rate. However, in Table 4 we display results for our synthesis when there are no resource constraints, showing an increase of up to 41.4% reduction, with FFT doubling its latency reduction. This shows if that we could explore different ways of resource sharing and binding, our results could be greatly improved.

## B. Validation on Cyclone II (2C35) FPGA Board Results

To validate the effectiveness of our architecture, we used our BTW scheduler to implement the DES benchmark on Altera's Cyclone II (2C35) FPGA board. We set a 1 clock cycle worst-case delay for a 32-bit adder, and 2 clock cycle worst-case delay for a 16x16 bit multiplier, and targeted a 1 clock cycle completion time (introducing timing yield) for the multiplier. For the purpose of this

design, we did not allow resource sharing, and assumed there were no pipeline operations. Therefore, we simplified the Razor register in the variation-tolerant architecture into the recovery register shown in Figure 5a. The timing diagram of the register when fault is presented is shown in Figure 5b.

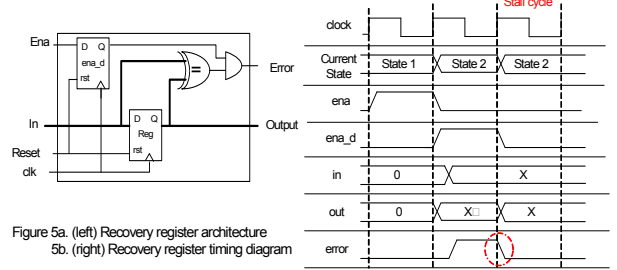


Figure 5a. (left) Recovery register architecture  
5b. (right) Recovery register timing diagram

The scheduling result from the BTW scheduler is shown in Figure 6. We then used Altera's Quartus II v 6.0 to conduct RTL synthesis and physical design. The timing analysis tool from Quartus II gave us a worst-case delay of 15.06 ns. To test the design, we used a random set of 2048 input vectors and clocked the design at better than worst-case frequencies. Table 5 lists the clock periods that we allowed the design to use and the total latency needed to run through the set of input vectors and produce the correct results. As can be observed from this table, an increase in frequency causes a more frequent detection and correction of faults. However, at a certain point, as we are running at a higher frequency, the overall latency is reduced even though extra stall cycles are introduced to recover fault. Our architecture shows an encouraging result of at most nearly 43.5% latency reduction compared to using the worst-case analysis from the Quartus timing analysis (row 1 in Figure 5). This shows that our architecture will indeed allow us to efficiently perform at a better than worst-case frequency.

However, one thing worth noting is that we were able to clock the design all the way down to about 9.6 ns before faults were detected. This can most likely be attributed to the following:

1. Chip manufactures usually overcome process variation by selling chips at different speed grades. It is very likely that the board is one of the higher speed grades, making the worst-case analysis overly conservative.
2. The worst-case analysis considers extreme environmental settings, like high temperature, supply voltage fluctuation, etc., while these scenarios do not occur in our experiments.
3. We only performed tests on 2048 different input vectors out of the millions of possible combinations. It is very likely that the multiplier was able to complete the operations without causing switching in the later stages of the cycle.

Even when comparing with the optimistic assumption that the circuit in our experiment can be clocked at 9.6ns without introducing faults, the mechanism of fault detection and correction provides about 9.5% latency reduction. In

practice, determining the tight bound for worst-case delay for each individual chip can be nontrivial and requires a lot of testing; thus the ability to dynamically provide higher performance when faults do not occur very often is desirable.

## 6. Conclusion and Future Work

In this paper we presented an ILP-based BTW scheduler that caters to the specific need of variation-tolerant architectures, and the experiment results demonstrate the effectiveness of better than worst-case designs. However, the ILP scheduling solution is known to be difficult for scaling to large problem sizes. We are continuing to explore opportunities for more scalable solutions. One possible direction is to use LP approximation, and then use the ordering of the scheduling variables as inputs to more scalable heuristics, including the use of the SDC scheduling solution [9]. We also plan to extend the BTW scheduling problem to CDFGs and develop efficient solutions. Another degree of freedom we may explore in scheduling for the stallable-FSM architecture is clock period selection, as the clock period is no longer constrained by the longest path of operations scheduled in each clock cycle. With the delay distribution of each function unit, clock period selection will directly affect the error probability of each operation, which can significantly affect the outcome of the total expected latency, as shown in Section 5b. Therefore, we are looking to include clock selection in our HLS flow to minimize the total expected latency of the circuit.

Table 3. BTW scheduler results with resource constraints

	#Op	BTW	BTW-exp	WC	Improve	Time(s)
DES	11	10	10.66184	14	23.84%	0.03
FFT	27	16	16.5136	19	13.07%	128.2
AR	28	22	23.32	28	16.71%	0.05
FIR16	31	23	24.272	35	30.65%	1.28
EWf	35	15	16.1	22	26.82%	0.67

Table 4. BTW scheduler results without resource constraints

	#Op	BTW	BTW-exp	WC	Improve	Time(s)
DES	11	10	10.50131	14	24.99%	0.61
FFT	27	11	11.57	16	27.69%	626.3
AR	28	17	18.79	25	24.84%	450.5
FIR16	31	19	20.6536	35	40.99%	955.6
EWf	35	11	12.892	22	41.4%	0.86

Table 5. DES board example results

Clk Period	faults	latency (cc)	latency (us)	Reduction
15.02 (ns)	0	8192	123.04384	-
10	0	8192	81.92	33.42%
9.6	0	8192	78.6432	36.09%
9.41	1	8193	77.09613	37.34%
9.23	4	8196	75.64908	38.52%
9.09	8	8200	74.538	39.42%
9	19	8211	73.899	39.94%
8.88	26	8218	72.97584	40.69%
8.69	36	8228	71.50132	41.89%
8.57	40	8232	70.54824	42.66%
8.42	59	8251	69.47342	43.54%

8.33	395	8587	71.52971	41.87%
8.18	602	8794	71.93492	41.54%
8	2053	10245	81.96	33.39%

## Acknowledgement

Financial supports from the National Science Foundation (US) under award CCF-0530261, the Semiconductor Research Corporation (SRC) under Contract 1879.001, and the Natural Science Foundation of China under award 60728205 are greatly acknowledged.

## References

- [1] Altera DE2 Board, "http://www.altera.com/education/univ/materials/boards/unv-de2-board.html#boardinfo"
- [2] S. Nassif, V. Pitchumani, N. Rodriguez, D. Sylvester, C. Bittlestone, R. Radojicic, "Variation-Aware Analysis: Savior of the Nanometer Era?" DAC, pp. 411-412, 2006.
- [3] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xPilot: A Platform-Based Behavioral Synthesis System," SRC TechCon, 2005.
- [4] J. Jung, T. Kim, "Timing Variation-Aware High-Level Synthesis", Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design, p. 424-428, 2007, San Jose, California, United States.
- [5] F. Wang, G. Sun, Y. Xie, "A Variation Aware High Level Synthesis Framework," *Design, Automation and Test in Europe, 2008. DATE '08*, vol., no., pp.1063-1068, 10-14 March 2008
- [6] W.L. Hung, X. Wu, Y. Xie, "Guaranteeing Performance Yield in High-Level Synthesis," *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, vol., no., pp.303-309, 5-9 Nov. 2006
- [7] J. Cong, 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.
- [8] S. Suhaib, D. Mathaikutty, S. Shukla, "Dataflow Architectures for GALS," *Electron. Notes Theor. Comput. Sci.* 200, 1 (Feb. 2008), 33-50.
- [9] J. Cong, Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," *Design Automation Conference, 2006 43rd ACM/IEEE*, vol., no., pp.433-438, 0-0
- [10] 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
- [11] D. D Gajski, N.D. Dutt, A.C. Wu, S.Y. Lin, "High Level Synthesis: Introduction to Chip and System Design", 1992; Kluwer Academic
- [12] Z. Wo; I. Koren, M. Ciesielski, "An ILP formulation for yield-driven architectural synthesis," *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*, vol., no., pp. 12-20, 3-5 Oct. 2005
- [13] S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, W-L. Hung, "An ILP Formulation for Reliability-Oriented High-Level Synthesis", Proceedings of the 6th International Symposium on Quality of Electronic Design, p.364-369, March 21-23, 2005
- [14] Y. Chen, J. Ouyang, Y. Xie, "ILP-based scheme for timing variation-aware scheduling and resource binding," *SOC Conference, 2008 IEEE International*, vol., no., pp.27-30, 17-20 Sept. 2008

- [15] S. Chaudhuri, R.A. Walker, "ILP-based scheduling with time and resource constraints in high level synthesis," *VLSI Design, 1994., Proceedings of the Seventh International Conference on* , vol., no., pp.17-20, 5-8 Jan 1994
- [16] W.T. Shiue, C. Chakrabarti, "ILP-based scheme for low power scheduling and resource binding," *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on* , vol.3, no., pp.279-282 vol.3, 2000
- [17] T. Austin, V. Bertacco, D. Blaauw, T. Mudge, "Opportunities and challenges for better than worst-case design," *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific* , vol.1, no., pp. 1/2-1/7 Vol. 1, 18-21 Jan. 2005
- [18] A.J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder", in *Formal Methods in System Design*, volume 1:1, July 1992, pp. 119–137.
- [19] M.E. Dean, D.L. Dill and M. Horowitz, "Self-Timed Logic Using Current-Sensing Completion Detection", in *Proceedings of ICCD*, 1991.
- [20] S.M. Nowick, K.Y. Yun, A.E. Dooply and P.A. Beerel, "Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders," in Proc. ASYNC, 1997.