Recovery-Based Resilient Latency-Insensitive Systems

Yuankai Chen†, Xuan Zeng‡, and Hai Zhou†
† EECS Department, Northwestern University, Evanston, IL, U.S.A.
‡ Microelectronics Department, Fudan University, China

***Abstract*—As the interconnect delay is becoming a larger fraction of the clock cycle time, the conventional global stalling mechanism, which is used to correct error in general synchronous circuits, would be no longer feasible because of the expensive timing cost for the stalling signal to travel across the circuit. In this paper, we propose recovery-based resilient latency-insensitive systems (RLISs) that efficiently integrate error-recovery techniques with latency-insensitive design to replace the global stalling. We first demonstrate a baseline RLIS as the motivation of our work that uses additional output buffer which guarantees that only correct data can enter the output channel. However this baseline RLIS suffers from performance degradations even when errors do not occur. We propose a novel improved RLIS that allows erroneous data to propagate in the system. Equipped with improved queues that prevent accumulation of erroneous data, the improved RLIS retains the system performance. We provide theoretical study that analyzes the impact of errors on system performance and the queue sizing problem. We also theoretically prove that the improved RLIS performs no worse than the global stalling mechanism. Experimental results show that the improved RLIS has 40.3% and even 3.1% throughput improvements compared to the baseline RLIS and the infeasible global stalling mechanism respectively, with less than 10% hardware overhead.**

I. Introduction

The evolution of the silicon industry over past decades has been fueled by continued scaling. However, in contrast to the increasing transistor speed, the interconnect speed suffers from the reduced wire cross-sectional area and the conductor spacing [1]. This issue exacerbates due to the ever-increasing operating frequency, chip size and average interconnect length. The interconnect delay is becoming a larger fraction of the clock cycle time, and some global wires may need to be pipelined into multiple clock cycles [2].

Latency-insensitive system (LIS) [3] has been proposed in recent years to relieve the designers of concerning interconnect delays in early design stage. An LIS can be derived from the original synchronous system, by partitioning the system into individual modules and encapsulating each module with a shell that implements the latency-insensitive protocol. Any number of wire-pipelining registers (relay stations) can be inserted between modules without affecting circuit functionality. More details about LIS will be reviewed in Section III.

Runtime errors are also another critical design concern in the modern silicon industry. With aggressive timing designs and the faster transition speed, effects such as power-voltage-temperature (PVT) variations and alpha particle radiation would cause the circuit behavior no long deterministic. More explicitly, PVT variations cause fluctuation in circuit speed, which leads to timing violation. If the alpha particle radiation hits a register at its transition time, an erroneous value may be stored and as a consequence occurs a transient error.

Recovery-based principle is widely adopted in many existing techniques to enable the circuit to correct runtime errors, such as Razor [4] and TRC [5]. The general idea of the error-recovery technique is that once an error is detected, the corresponding component takes additional clock cycles to recompute the correct data. Traditionally, in order to maintain the computation sequence, applying error-recovery techniques to *general* circuits requires stalling the whole circuit to wait for the errant component to recover from its error. In the rest of this paper, we refer such mechanism as *global stalling*. In the global stalling mechanism, the error signals must be collected within a clock cycle time, and also the stalling signals must reach each component in the same clock cycle. This global stalling mechanism is infeasible in modern circuits where the interconnects take different number of clock cycles, as it cannot guarantee that the stalling across the whole circuit is synchronized within one clock cycle, whereas asynchronous stalling may lead to a disorder of computation sequence.

Latency-insensitive protocol in the LIS provides non-deterministic nature to the system: the computation sequence can be flexible as each module carries out its computation independently of the others. We believe that this non-deterministic nature can be exploited to avoid the global stalling if the strict computation sequence can be relaxed. In this work, we consider extending LIS with error-recovery techniques to overcome the fore-mentioned drawback of the global stalling mechanism within a single framework, resilient latency-insensitive system (RLIS). The contributions of our work are as follows:

- We first demonstrate a baseline RLIS that is abstracted from an existing technique [6]. The baseline RLIS uses additional output buffer such that only validated data is allowed to enter the output channel. An analysis is presented to show that the baseline RLIS suffers from performance degradation even when errors do not occur.
- We propose a novel improved RLIS where erroneous data is allowed to propagate in the system, but with a "signature bit" erroneous data can be distinguished from correct data. The error correction in the improved RLIS is not confined to an individual module, but it is done by the whole system collectively in a more efficient manner. We design improved queues to prevent erroneous data from accumulating in the system.

- We present thorough theoretical analysis of the improved RLIS. We study the impact of erroneous data on the system performance and the queue sizing problem. We compare the improved RLIS with the global stalling mechanism, and conclude that the improved RLIS performs no worse than the infeasible global stalling mechanism.

The rest of this paper is organized as follows. Section II summarizes existing related work on resilient designs and LIS. Section III introduces assumptions and useful background concepts in our model. Section IV describes the baseline RLIS as the motivation of our work. Section V presents our improved RLIS design and its theoretical analysis. Experimental results are shown in section VI in terms of the system performance and the hardware overhead. The work is concluded in section VII.

II. Related Work

Error resiliency have been studied at different design levels. For instance, at circuit level, S. Mitra proposed a c-element based circuitry to eliminate radiation-induced soft errors at register level [7]. Razor [4] and TRC [5] are architectural level approaches that employ additional circuitries to detect timing errors. ANT [8] is a system level approach that permits error to occur in a signal processing block and later on corrects it via an error correction block constructed based on case-by-case computation properties. It is worth mentioning that [6] proposes "recovery islands" that buffer outputs and enable different components of SoC to recover separately. Although the context of their work is different from ours and the communication is implemented with system bus, their idea can be abstracted as an error-recovery methodology, which is taken as a baseline RLIS and the motivation of our work.

Component-based design and optimization, where the system is modeled as a set of interconnecting modules, have been prevailing to cope with the ever-increasing complexity in modern circuits [9]. One of those models is LIS, first proposed by L. P. Carloni et al. in [3], as a design methodology which effectively overcomes interconnect-sensitive issues induced by long wires among modules. Theoretical analysis of LIS was originally presented in [10], based on the assumption of infinite queue sizes. Several works followed to consider the impact of finite queue sizes on the system performance, and propose algorithms to solve the queue sizing problem. In particular, R. Lu and C. K. Koh formulated the queue sizing problem as a mixed integer linear programming problem (Mixed ILP) [11]. R. Collins et al. [12] proposed marked graph model for analyzing the relationship between queue sizes and system performance. They studied the optimal queue sizes for different graph topologies and proved that the queue sizing problem for general topology is NP-complete. A heuristic algorithm was also proposed to solve the problem efficiently. In a recent work [13], J. D. Huang et al. proposed compacted quantitative graph (CQG) to represent original LIS graph, where the number of nodes and edges are reduced by path condensation and edge unification. ILP is performed on CQG to find the optimal queue sizes. Their approach is able to find optimal solution with a smaller size of ILP.

III. Preliminaries: Models, Background and Assumptions

A. Error-Recovery Model

We define critical modules as the modules where runtime error would occur with a certain rate. The critical modules are equipped with certain error detection circuitry. The circuitry is able to notify the module at the early stage of every clock cycle that if any error occurred during the last computation. If an error is detected, recomputation should be carried out immediately, and the new computation is delayed until the correct output of the last computation is produced. We believe that this model assumption is general enough and applicable to many error-recovery techniques such as Razor and circuit replica. Note that, although Razor does not need physical "recomputations", it matches our model because it does take additional clock cycle to produce the correct output.

B. Latency Insensitive System

An LIS consists of a set of computation modules and a set of channels that connect the modules and deliver data between them. Each module is encapsulated with a shell that implements latency-insensitive protocol. The latency-insensitive protocol ensures that the module only carries out the computation when all of its inputs are available. The shell will clock-gate the module when some of its input channels do not have data ready or a stalling request is received.

Channel is basically a queue of buffers that delivers data from its producer to its consumer. In practice the queue size is finite, therefore when a queue is full, the queue issues a stall signal to its source module to avoid overflow. If the interconnect is longer than a clock cycle time, pipelining registers, referred as relay stations, are inserted into the channel. The relay stations are initialized with void data τ.

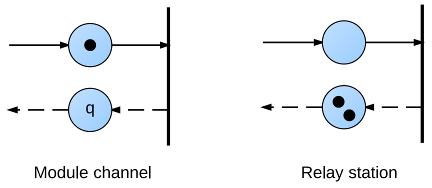

Fig. 1. Marked graph representation of LIS.

The performance of LIS is usually measured by its throughput. Marked graph [12] is used to analyze the throughput (Fig. 1). Each channel is transformed into a forward edge and a backward edge. Modules and relay stations are transformed into transitions (vertical bars in Fig. 1). For each incoming channel of a module, the initial number of tokens on the forward edge is one, and it is equal to the channel queue size on the backward edge. For each incoming channel of a relay station, the forward edge has no initial tokens, and the backward edge has two tokens initially. The throughput of a given LIS is equal to the minimum cycle mean of the marked graph. The mean of a cycle in the marked graph is the number of tokens in the cycle divided by the number of edges.

IV. Motivation: Baseline Resilient LIS

We first introduce a baseline RLIS design that is abstracted from the recovery islands proposed in [6]. In the baseline RLIS, every output of critical modules is buffered for one additional clock cycle for validation. Only validated data are allowed to enter the channel and be delivered between modules. Our analysis will show that the baseline RLIS suffers from a performance degradation caused by the additional buffer if the system graph contains cycles.

Fig. 2 illustrates the baseline RLIS. For each critical module, in addition to the shell, a buffer is inserted in between the shell and the output channel. Every output datum will first be stored in the buffer for one clock cycle. If no error is reported in the next cycle, the datum is pushed to the output channel. Otherwise, $recompute$ signal triggers recomputation in module, the erroneous datum is discarded and a bubble is pushed to the output channel instead. It guarantees that every datum sent to the communication channel is correct. Obviously, the system retains correctness-by-construction property.

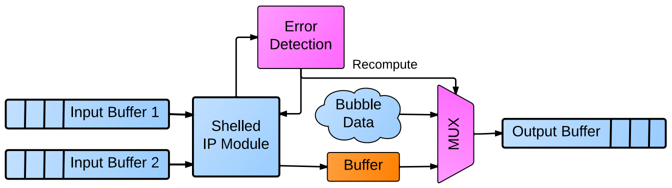

Fig. 2. A module in the baseline RLIS.

However, the baseline RLIS suffers from a performance degradation even when error does not occur. An example system is shown in Fig. 3(a), where the system is composed of three modules and three channels. Suppose that module B is a critical module. With baseline RLIS wrapping, the equivalent LIS is shown in Fig. 3(b). Note that the buffer between the shell and output channel CH_2 of module B in the baseline RLIS design is equivalent to inserting an additional relay station between B and C. With four channels and one relay station, the throughput of the system is decreased to 3/4, from original 1.0.

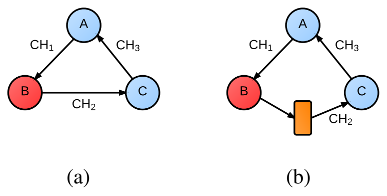

Fig. 3. Illustration of performance degradation. (a) Original LIS. (b) Equivalence of baseline RLIS.

V. Improved Resilient LIS

In this section, we propose an improved RLIS with an improved queue mechanism that overcomes the performance degradation of the baseline RLIS. A thorough theoretical study of the improved RLIS will be presented, which studies the queue sizing problem in improved RLIS, and makes theoretical comparison with the global stalling mechanism.

A. Design

The performance degradation in the baseline RLIS mainly comes from the conservative strategy that requires all data entering the channel be correct. As a consequence, correct data also need to be buffered additionally for one clock cycle.

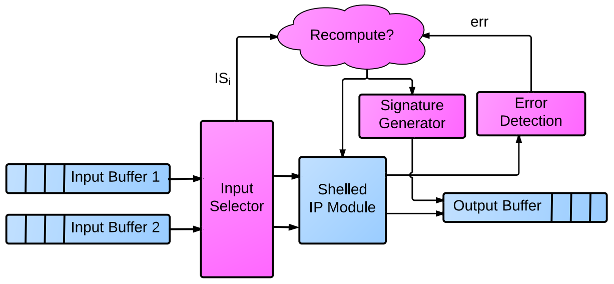

Fig. 4. A module in the improved RLIS.

Fig. 4 illustrates the improved RLIS. The additional output buffer being removed from the system means that every produced datum enters the output channel. In order to distinguish correct data and erroneous data, each module attaches a "signature bit (sig_bit)" to every datum that it produces. The signature bit only flops when a new computation is carried out. In other words, in the events of recomputation, the signature bit remains the same. In hardware, the signature bit can be easily generated with a $recompute$ signal and a flip-flop recording its previous value (sig_bit' is the value of the signature bit of the previous clock cycle):

$$sig\_bit = recompute \textbf{ xnor } sig\_bit' \tag{1}$$

In the improved LIS, the recomputation is not required only when an error is detected in the computation, but also when the module receives erroneous input data. Erroneous input data can be easily detected by checking their signature bits. If two consecutive data received from the same input channel possess the same signature bit, the first datum must be erroneous and a recomputation is required. Therefore, the recompute signal in the improved RLIS can be expressed as:

$$recompute = err \lor IS_1 \lor IS_2 \lor \cdots \lor IS_n \tag{2}$$

where err is the signal indicating whether error is detected from the computation of the previous cycle, and IS_i is the signal generated from input channel i, indicating whether the datum at the head of the channel has the same signature bit as the previous received one:

$$IS_i = sig\_bit_i \textbf{ xnor } sig\_bit_i' \tag{3}$$

The inputs to the module computation are first processed by an input selector. For every channel, the input selector has additional register to store the datum used in the previous clock cycle. Each input to the module is selected from two sources: the datum at the head of the input channel and the previous datum. When $recompute$ is zero, the system works the same as the original LIS. If $recompute$ is one and some IS_i equals to one, it means that the recomputation is needed because the previous datum of channel i is erroneous, therefore the datum at the head of the channel is popped-out and used in the recomputation. Otherwise, the previous datum is used and datum does not pop out from channel i.

With allowing erroneous data entering system channels, the erroneous data may accumulate in the system, taking up buffers and degrading the system performance. It happens when the erroneous datum enters a cycle, it will circulate in the cycle and never get out. We design an "improved queue" to resolve this issue in the improved RLIS. Because every series of erroneous data will always be followed by a correct datum which possesses the same signature bit as the previous erroneous data, the basic idea of the improved queue is called "data replacement": when a datum enters a channel, the previous datum in the channel will be replaced if it possesses the same signature bit as the incoming datum. The storage element of the improved queue is illustrated in Fig. 5. The XNOR gate compares the signature bits of the incoming datum and its stored datum. If the output of XNOR gate is true (signature bits are the same), and the element is the tail of the queue (isEnd is true), it replaces the stored datum with the incoming datum. In other cases, the improved queue works the same as normal queues.

The effect of the improved queue on the system performance will be presented in the next subsection. An immediate observation can be made is that, with the data replacement function, each channel will contain at most one erroneous datum, preventing the unlimited accumulation of erroneous data.

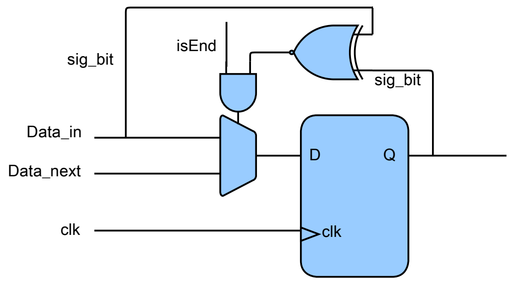

Fig. 5. Storage element of the improved queue.

B. Theoretical Analysis

In this subsection we analyze the performance of the improved RLIS. We will prove that the system will restore to its original performance eventually, therefore there is no need to adjust queue sizes for erroneous data. At the end, we will prove that the improved RLIS performs no worse than the global stalling mechanism.

We employ the marked graph to analyze the throughput of the system. In the improved RLIS, the number of tokens in a cycle can only change when one of the following two events happens:

- **Channel stuttering**: We define channel stuttering as the event that the sink module of a channel does not consume data in the channel. It happens when the sink module carries out recomputation and the channel does not hold erroneous datum at its head. The effect on token changes is illustrated in Fig. 6. In the circles are the actual changes when this event happens, and in the parenthesis are the changes in the normal situation. The forward edge has one more token than usual, while the backward edge has one less. This results that any cycle that contains the forward edge has one more token, and any cycle that contains the backward edge has one less token.
- **Data replacement**: Data replacement happens when the incoming datum replaces the previous datum in the improved queue. The effect on token changes is also illustrated in Fig. 6. The forward edge has one less token than usual, while the backward edge has one more. This results that any cycle that contains the forward edge has one less token, and any cycle that contains the backward edge has one more token. Data replacement can be viewed as the reversed operation of channel stuttering.

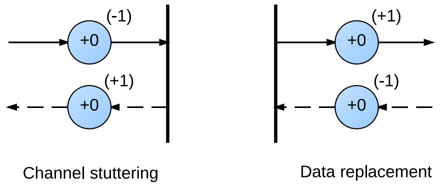

Fig. 6. Illustration of token changes of channel stuttering and data replacement.

There are three types of cycles in the marked graph: forward cycles that contain forward edges only, backward cycles that contain backward edges only and mixed cycles that contain both forward and backward edges. We will examine the throughput of each of these three types of cycles in the improved RLIS.

First we examine a forward cycle C with m channels. Because tokens on forward edges represent actual data, the tokens in C represent either correct data (correct tokens) or erroneous data (erroneous tokens). Suppose the number of correct tokens is n_1 and the number of erroneous tokens is n_2. We consider the following two cases:

- $n_1 > m$: There is at least a channel that has more than one token. Any erroneous datum in C will travel to that channel. When an erroneous datum enters the channel, and because the following datum must have the same signature, the erroneous datum will be replaced when the next datum enters. Therefore any erroneous data in C will be eventually eliminated. In this case, the throughput of C restores eventually.
- $n_1 \leq m$: Cycle C will eventually have $n_1 + n_2 \leq m$, otherwise, the extra erroneous token will be replaced by the improved queue as in the previous case. Since processing erroneous data does not count for throughput, the throughput of C retains n_1/m.

Backward cycles have the same throughputs as their corresponding forward cycles, because they share the same set of transitions (modules). Therefore their throughputs will eventually restore too.

Mixed cycles in the marked graph represent reconvergent paths in the original system graph. We consider a backward edge where channel stuttering just happened. As a result, the number of tokens of the cycle decreases by one. Because it is a mixed cycle, going backward along backward edges one will eventually reach a reconvergent node where the next edge is forward edge. Taking Fig. 7 for example, where a reconvergent node is depicted, consider the mixed cycle containing backward edge of CH_1 and forward edge of CH_2. Suppose channel stuttering has happened on a channel in the same cycle and in the upstream of CH_1. There are three scenarios:

- If data replacement has happened before the resulted erroneous data reaches the reconvergent node, the number of tokens in this cycle has restored to its original value.
- If no data replacement has happened, and when the resulted erroneous data reaches the reconvergent node, CH_2 holds a correct datum at its head, channel stuttering will occur on CH_2 and cause the number of tokens on the forward edge of CH_2 one more than usual, which therefore increases and restores the number of tokens in the cycle.
- If no data replacement has happened, and when the resulted erroneous datum reaches the reconvergent node, CH_2 holds an erroneous data at its head, it means that another channel stuttering has occurred in the upstream of CH_2 in the cycle. There must exist the a forward edge in the upstream of CH_2 in the cycle where the channel stuttering occurred, which has already restored the number of tokens of the cycle. Note that it is impossible for the channel stuttering to occur on a upstream channel with a backward edge in the cycle, since erroneous data cannot pass from that channel to a downstream channel with a forward edge in the cycle.

We have proved that in each of these three scenarios, the number of tokens in the cycle will eventually restore. The same reasoning can be applied to a cycle that contains a forward edge where channel stuttering just happened. Therefore the number of tokens in mixed cycles also restores to its original value.

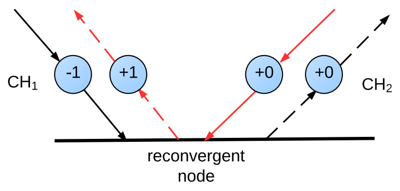

Fig. 7. Reconvergent node and channel stuttering happens on CH_2 when the erroneous data arrives on CH_1.

Based on the above analysis, we can make the following claim:

***Claim** 1:* The performance of the improved RLIS will eventually restore to the performance before error occurrence.

As the performance of improved RLIS is not degraded by erroneous data, there is no need to adjust queue sizes:

***Claim** 2:* Optimal queue sizing of original LIS without considering erroneous data retains optimal in improved RLIS.

Last we make the comparison between the improved RLIS and global stalling.

***Claim** 3:* The performance of improved RLIS is no worse than global stalling system.

Proof: In the global stalling system, once an error occurs, the whole circuit is stalled for one clock cycle and recomputation is carried out on those errant parts. Therefore we are going to prove that in the improved RLIS, each module at most "wastes" one clock cycle when an error occurs.

In the improved RLIS, suppose module A produces an erroneous datum. First we consider the modules in the same strongly connected component (SCC) with A. If that erroneous data will be eliminated eventually, it will goes through each module at most once before it enters a channel with more than two tokens. Therefore each module in the SCC processes the erroneous datum at most once, which is equivalent to being stalled for a clock cycle. If that erroneous datum stays in a cycle forever, it actually takes the place of a bubble and it does not affect the system performance.

Second, we consider other modules which are not in the same SCC with A. Since the graph is a DAG after contracting each SCC into a single node, the erroneous datum will enter other succeeding SCCs at most once. As the same as the above analysis, the effect on succeeding SCCs is equivalent to stalling each module in those SCCs for a clock cycle. The preceding SCCs of A may be affected by back-pressure effect (some channel queue in between is full). Because of the DAG structure, back-pressure will enter those SCCs at most once too. We conclude that each module in the improved RLIS will be affected for at most one clock cycle. ■

VI. Experimental Results

A. Performance Evaluation

We simulate three systems and compare their performances: baseline RLIS, improved RLIS and global stalling. Systems are implemented in C++ on Linux platform. We first simulate the systems on a real-world design, MPEG encoder, whose system graph is adopted from [10] with relay stations are inserted on channel a_{13}, a_{15} and a_{19}. We treat every module as critical module except the source and the sink. To test the systems on larger graphs with higher complexity, we also randomly generated some system graphs. In each graph, we randomly pick 5% of the modules as critical modules. A random error rate between 1e-4 and 5e-4 (probability per cycle) is assigned to each critical module. The throughput of the system is calculated by dividing the number of correct data processed by the sink node by the number of total clock cycles. We implemented heuristic algorithm in [12] to assign queue sizes to channels.

Table I shows the results of the simulation. We simulated 1 million cycles for each test case. In the table, the number of modules, the number of channels, the number of critical modules and the original throughput without errors are listed to reflect graph structures. We list the throughputs of the three systems and compare the improved RLIS with the other two. From the table we can see that the improved RLIS has 40.3% performance improvement over baseline RLIS on average, and it performs 3.1% better than global stalling on average, as expected from Claim 3.

TABLE I
SIMULATION RESULTS OF BASELINE RLIS, IMPROVED RLIS AND GLOBAL STALLING.

Tests	Graph Properties				Throughputs			Improvements of improved RLIS	
	#Modules	#Channels	#Critical Modules	Ori. Throughput	Baseline(TP_{bs})	Improved(TP_{im})	Global Stalling (TP_{gs})	$\frac{TP_{im}-TP_{bs}}{TP_{bs}}$	$\frac{TP_{im}-TP_{gs}}{TP_{gs}}$
MPEG	17	22	15	0.875	0.452	0.820	0.800	81.4%	2.5%
T50	50	250	9	0.75	0.498	0.745	0.735	49.5%	1.3%
T100	100	1000	23	0.6	0.485	0.59	0.575	21.6%	2.6%
T150	150	2250	32	0.529	0.386	0.525	0.503	36.0%	4.3%
T200	200	4000	50	0.428	0.373	0.421	0.403	12.8%	4.4%
Average								40.3%	3.1%

B. Area Evaluation

We implemented the improved RLIS in RTL to measure its area overhead against the original LIS. The overhead is compared in two parts: the overhead of the improved queue and the overhead of the module wrapping. In the queue comparison, we implemented an original queue of 8 elements and each element is 8-bit in width. Therefore the improved queue has the same size, but each element has 9 bits, including the signature bit. In the module wrapping comparison, we wrapped a multiplier module chosen from an MPEG decoder implementation [14]. The multiplier takes two 16-bit numbers and a 2-bit $code$ as inputs. The module selects two IDCT coefficients depending on $code$, and multiplies the input numbers by the two IDCT coefficients respectively. The products are the outputs of the module. Because IDCT coefficients are constant, the multiplication is decomposed into a set of shifting and adding operations and completed within one clock cycle. We wrapped the module for both the original LIS and the improved RLIS. We assume that the improved RLIS is resilient against timing errors caused by PVT variations, and thus we integrated Razor flip-flop [4] to detect errors.

TABLE II
AREA OVERHEAD OF IMPROVED RLIS. THE UNIT OF AREA IS μm^2

	Original LIS	Improved RLIS	Overhead
Queue	551.45	597.34	8.32%
Multiplier	1465.60	1579.40	7.76%

Table II shows the area overheads of the improved RLIS. The area is measured by Synopsys Design Compiler with 45nm technology node. The overheads are both small: the improved queue has an overhead of 8.32% and the module wrapping has an overhead of 7.76%. The overhead of the improved queue mostly comes from the additional signature bit. The overhead of the module wrapping depends on the complexity of the original module. The multiplier is a relatively simple module. For modules with the same sizes of inputs and outputs but more complex computation, we expect the the overhead would be relatively smaller.

VII. CONCLUSION

In this work, we proposed recovery-based resilient latency-insensitive system that extends latency-insensitive design with error-recovery technique to overcome the infeasibility of traditional global stalling mechanism in modern circuits. We were motivated by a baseline RLIS design with additional output buffers that suffers from severe performance degradation issues. We proposed an improved RLIS with improved queues that retains the system performance. We provided theoretical studies that analyze the queue sizing problem and system throughput. Experimental results show that the improved RLIS has 40.3% and 3.1% throughput improvements compared to the baseline RLIS and the global stalling mechanism respectively, with less than 10% hardware overhead.

VIII. ACKNOWLEDGEMENT

This work is supported by NSF CCF-1115550 and NSFC-61228401.

REFERENCES

[1] R. Havemann and J. Hutchby, “High-performance interconnects: an integration overview,” *Proceedings of the IEEE*, vol. 89, no. 5, pp. 586 –601, may 2001.
[2] M. Bohr, “Silicon trends and limits for advanced microprocessors,” *Commun. ACM*, vol. 41, no. 3, pp. 80–87, Mar. 1998.
[3] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, “A methodology for correct-by-construction latency insensitive design,” in *IEEE Proc. ICCAD*, 1999, pp. 309 –315.
[4] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, “Razor: circuit-level correction of timing errors for low-power operation,” *Micro, IEEE*, vol. 24, no. 6, pp. 10 –20, nov.-dec. 2004.
[5] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, “Circuit techniques for dynamic variation tolerance,” in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC ’09, 2009, pp. 4–7.
[6] V. Kozhikkottu, S. Dey, and A. Raghunathan, “Recovery-based design for variation-tolerant SoCs,” in *IEEE Proc. DAC*, 2012, pp. 826–833.
[7] S. Mitra, M. Zhang, N. Seifert, T. Mak, and K. S. Kim, “Soft error resilient system design through error correction,” in *Very Large Scale Integration, 2006 IFIP International Conference on*, oct. 2006, pp. 332 –337.
[8] N. Shanbhag, “Reliable and energy-efficient digital signal processing,” in *Proceedings of the 39th annual Design Automation Conference*, ser. DAC ’02, 2002, pp. 830–835.
[9] Y. Lu and H. Zhou, “Efficient design space exploration for component-based system design,” *IEEE Proc. ICCAD*, 2012.
[10] L. P. Carloni and A. L. Sangiovanni-Vincentelli, “Performance analysis and optimization of latency insensitive systems,” in *IEEE Proc. DAC*, 2000, pp. 361–367.
[11] R. Lu and C.-K. Koh, “Performance optimization of latency insensitive systems through buffer queue sizing of communication channels,” in *IEEE Proc. ICCAD*, 2003, pp. 227–231.
[12] R. Collins and L. Carloni, “Topology-based performance analysis and optimization of latency-insensitive systems,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 12, pp. 2277 –2290, dec. 2008.
[13] J.-D. Huang, Y.-H. Chen, and Y.-C. Ho, “Throughput optimization for latency-insensitive system with minimal queue insertion,” in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, jan. 2011, pp. 585 –590.
[14] MPEG decoder, http://www.ece.mcmaster.ca/$\sim$nicola/mpeg.html.