Hindawi
International Journal of Reconfigurable Computing
Volume 2017, Article ID 5419767, 17 pages
https://doi.org/10.1155/2017/5419767

Research Article

A High-Level Synthesis Scheduling and Binding Heuristic for FPGA Fault Tolerance

David Wilson,[1] Aniruddha Shastri,[2] and Greg Stitt[1]

[1]*Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA*
[2]*National Instruments Corp., 11500 N Mopac Expwy, Austin, TX 78759, USA*

Correspondence should be addressed to David Wilson; d.wilson@ufl.edu

Received 31 March 2017; Revised 2 July 2017; Accepted 11 July 2017; Published 21 August 2017

Academic Editor: Michael Hübner

Computing systems with field-programmable gate arrays (FPGAs) often achieve fault tolerance in high-energy radiation environments via triple-modular redundancy (TMR) and configuration scrubbing. Although effective, TMR suffers from a 3x area overhead, which can be prohibitive for many embedded usage scenarios. Furthermore, this overhead is often worsened because TMR often has to be applied to existing register-transfer-level (RTL) code that designers created without considering the triplicated resource requirements. Although a designer could redesign the RTL code to reduce resources, modifying RTL schedules and resource allocations is a time-consuming and error-prone process. In this paper, we present a more transparent high-level synthesis approach that uses scheduling and binding to provide attractive tradeoffs between area, performance, and redundancy, while focusing on FPGA implementation considerations, such as resource realization costs, to produce more efficient architectures. Compared to TMR applied to existing RTL, our approach shows resource savings up to 80% with average resource savings of 34% and an average clock degradation of 6%. Compared to the previous approach, our approach shows resource savings up to 74% with average resource savings of 19% and an average heuristic execution time improvement of 96x.

1. Introduction

Recently, computing systems in space and other extreme environments with high-energy radiation (e.g., high-energy physics, high altitudes) have been turning to field-programmable gate arrays (FPGAs) to meet performance and power constraints not met by other computing technologies [1]. One challenge for FPGAs in these environments is susceptibility to radiation-induced single-event upsets (SEUs), which can alter the functionality of a design by changing bits in memories and flip-flops. Although radiation-hardened FPGAs exist, some of those devices are still susceptible to SEUs and commonly have prohibitive costs compared to commercial-off-the-shelf (COTS) devices [2].

To mitigate these issues, designers often use triple-modular redundancy (TMR) on COTS FPGAs. TMR is a well-known form of hardware redundancy that replicates a design into three independent modules with a voter at the outputs to detect and correct errors. Research has shown that TMR with frequent configuration scrubbing (i.e., reconfiguring faulty resources) provides an effective level of fault tolerance for many FPGA-based space applications [3].

One key disadvantage of TMR is the 3x resource overhead, which often requires large FPGAs that may exceed cost or power constraints for embedded systems. Although resource sharing is a common strategy for reducing this overhead, designers often apply TMR to register-transfer-level (RTL) code, where the productivity challenge of exploring resource sharing and scheduling options is often impractical. Furthermore, RTL code is not available for the common case of using encrypted or presynthesized IP cores, making such exploration impossible.

In this paper, we automate this exploration during high-level synthesis (HLS) by integrating resource sharing and TMR into a scheduling and binding heuristic called the *Force-Directed Fault-Tolerance-Aware* (FD-FTA) *heuristic* that provides attractive tradeoffs between performance, area, and redundancy. More specifically, the heuristic explores the

impact of varying hardware redundancy with the capability to correct an error, which we measure as an *error-correction percentage*. Our heuristic is motivated by the observation that, for many situations, error correction is not as critical as error detection. By allowing a designer to specify an error-correction percentage constraint that is appropriate for their application, our heuristic can explore numerous options between performance and area that would be impractical to do manually.

Although other FPGA work has also approached fault tolerance through high-level synthesis, that earlier work mainly focused on different fault models or reliability goals. For example, Golshan et al. [4] introduced an HLS approach for minimizing the impact of SEUs on the configuration stream, which is complementary to our approach. Shastri et al. [5] presented a conceptually similar HLS approach, but that work focused on minimizing the number of coarse-grained resources without considering their FPGA implementation costs, while also suffering from long execution times and the need for manual parameter tuning. This paper presents an extension of [5] that addresses previous limitations with an automated approach, showing resource savings of up to 74% compared to the earlier approach, while also reducing heuristic execution time by 96x on average.

Similarly, high-level synthesis for ASICs has introduced conceptually similar techniques [6], but whereas ASIC approaches must deal with transient errors, FPGAs must pay special attention to SEU in configuration memory that will remain until scrubbing or reconfiguration (referred to as *semipermanent errors* for simplicity). Due to these semipermanent errors, FPGA approaches require significantly different HLS strategies.

Compared to the common strategy of applying TMR to existing RTL code, our heuristic has average resource savings of 34% and displays significant improvements as the latency constraint and the benchmark size increases. For a latency constraint of 2x the minimum-possible latency, our heuristic shows average resource savings of 47%, which achieves a maximum of 80% resource savings in the largest benchmark.

The paper is organized as follows. Section 2 describes related work on studies pertaining to areas such as FPGA reliability and fault-tolerant HLS. Section 3 defines the problem and the assumptions of our fault model. Section 4 describes the approach and implementation of the FD-FTA heuristic. Section 5 explains the experiments used to evaluate the FD-FTA heuristic, and Section 6 presents conclusions from the study.

2. Related Work

With growing interest in FPGAs operating in extreme environments, especially within space systems, a number of studies have been done on assessing FPGA reliability in these environments. Some of these studies rely on analyzing FPGA reliability through models. For example, Ostler et al. [7] investigated the viability of SRAM-based FPGAs in Earth-orbit environments by presenting a reliability model for estimating mean time to failure (MTTF) of SRAM FPGA designs in specific orbits and orbital conditions. Similarly, Héron et al. [8] introduced an FPGA reliability model and presented a case study for its application on a XC2V3000 under a number of soft IP cores and benchmarks. In addition to reliability models, a number of studies have been done on emulating and simulating faults in FPGAs (e.g., [9, 10]). Rather than relying on costly testing in a radiation beam, such approaches facilitate cost-effective testing of fault-tolerant designs. Although many studies analyze SRAM-based technologies, other studies have also considered antifuse FPGAs and flash FPGAs. For example, McCollum compares the reliability of antifuse FPGAs to ASICs [11]. Wirthlin highlights the effects of radiation in all three types of FPGAs and explores the challenges of deploying FPGAs in extreme environments, such as in space systems and in high-energy physics experiments [2]. In this paper, we complement these earlier studies by presenting an HLS heuristic that transparently adds redundancy to improve the reliability of SRAM-based FPGA designs. As described in Section 3, the fault model for our heuristic makes several assumptions based on the findings of these earlier works.

In our approach, we leverage the use of TMR to apply fault tolerance to FPGA designs. TMR is a well-studied fault tolerance strategy in FPGAs that has been studied in different use cases and under additional modifications. Morgan et al. [12], for example, compared TMR with several alternative fault-tolerant designs in FPGAs and showed that TMR was the most cost-effective technique for increasing reliability in a LUT-based architecture. Other works have introduced modifications that are complementary to our TMR design and may be incorporated into our heuristic. For example, Bolchini et al. [13] present a reliability scheme that uses TMR for fault masking and partial reconfiguration for reconfiguring erroneous segments of the design. Our work focuses on automatically applying the TMR architecture from high-level synthesis and could incorporate partial reconfiguration regions for enhanced reliability at the cost of producing device-specific architectures.

Work by Johnson and Wirthlin [14] approaches the issue of voter placement for FPGA designs using TMR and compares three algorithms for automated voter insertion based on strongly connected component (SCC) decomposition. Compared to the naive approach of placing voters after every flip-flop, these algorithms are designed to insert fewer voters through feedback analysis. Since FPGA TMR designs consist of both applying redundancy and voter insertion, our paper focuses specifically on the problem of automated TMR and can be potentially used alongside these voter insertion algorithms.

Although scheduling and binding in high-level synthesis is a well-studied problem [15–17], many of those studies do not consider fault tolerance or error-correction percentage. More recent works have treated reliability as a primary concern in the high-level synthesis process but focus on different reliability goals in ASIC designs. For example, Tosun et al. [18] introduce a reliability-centric HLS approach that focuses on maximizing reliability under performance and area constraints using components with different reliability characterizations. Our work, in contrast, focuses on minimizing area under performance and reliability constraints

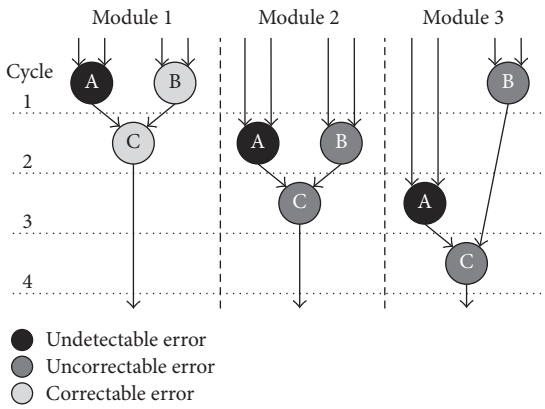

Figure 1: An example binding onto three color-coded resources. For this binding, a single fault can result in an undetectable, uncorrectable (but detectable), or correctable error, depending on the resource.

using components with the same reliability characterizations. Antola et al. [6] present an HLS heuristic that applies reliability by selectively replicating parts of the datapath for self-checking as an error-detection measure. Our heuristic differs by applying reliability through TMR with resource sharing. Our heuristic additionally varies the reliability through an error-correction percentage constraint that enables selective resource sharing between TMR modules.

Other work has notably used HLS to apply reliability through other means than replication. For example, Chen et al. [19] introduce an HLS approach that uses both TMR and gate-level hardening technique called gate sizing on different resources to minimize both soft error rate and area overhead. In another example, Hammouda et al. [20] propose a design flow that automatically generates on-chip monitors to enable runtime checking of control flow and I/O timing behavior errors in HLS hardware accelerators. Our heuristic focuses specifically on scheduling and binding and may be complementary to several of these approaches. Although our heuristic could also be potentially applied to ASIC design with modification, we have tailored scheduling and binding to the FPGA architecture, which notably has different HLS challenges compared to ASICs [21], and have compared its effectiveness with other FPGA-specific approaches. Our heuristic is especially applicable to FPGAs given the rise of FPGAs in space missions which commonly implement fault-tolerant logic through TMR and configuration scrubbing [3].

Compared to ASICs, far fewer reliability-centric high-level synthesis studies have targeted FPGAs. Golshan et al. [4] introduced a TMR-based HLS process for datapath synthesis, placement, and routing that targets SRAM-based FPGAs. Although conceptually similar to our work, that study focused on mitigating the impact of SEUs in the configuration bitstreams by enforcing self-containment of SEUs within a TMR module and by minimizing potential SEU-induced bridging faults that connect two separate nets in a routing resource. By contrast, our work focuses on minimizing the resources needed for TMR and can intentionally neglect self-containment of SEUs within a TMR module for additional resource savings. Dos Santos et al. [22] investigated another TMR-based HLS design flow on SRAM-based FPGAs and compares the reliability of these designs with their respective unhardened equivalents. Compared to our work, our heuristic focuses on making tradeoffs between redundancy and area, which can include full TMR. Shastri et al. [5] introduced a TMR-based HLS heuristic that focused on solely minimizing coarse-grained resources under latency and redundancy constraints. By contrast, our work considers each resource's implementation costs and uses improved scheduling and binding algorithms for better scalability in larger benchmarks and increased latency constraints. At a 2x normalized latency constraint, our heuristic provides average resource savings of 34% compared to TMR applied to existing RTL and achieves resource savings of 74% relative to [5] approach on the largest benchmark.

3. Problem Definition

Although there are different optimization goals that could be explored while varying the amount of error correction, in this paper we focus on the problem of *minimum-resource, latency- and error-constrained scheduling and binding*, which for brevity we simply refer to as *the problem*. To explain the problem, we introduce the following terms:

(i) Fault: a resource with an SEU-induced error

(ii) Module: one instance of the dataflow graph (DFG), analogous to a module in TMR

(iii) Error: any fault where one or more of the three modules output an incorrect value

(iv) Undetectable error: any fault where all three modules output the same incorrect value

(v) Detectable error: any fault where one or more modules output different values

(vi) Uncorrectable error: any fault where two or more modules output incorrect values

(vii) Correctable error: any fault where two or more modules output a correct value

(viii) Error-correction % (EC%): the percentage of total possible errors that are correctable by a given solution.

Figure 1 illustrates several example error types, where all operations with the same color are bound to a single resource. If the black resource experiences a fault, this binding results in an undetectable error because all three modules will produce the same incorrect value. If there is a fault in the medium-gray resource, this binding causes an uncorrectable error because two modules (2 and 3) will produce incorrect values. A fault in the light-gray resource results in a correctable error because modules 2 and 3 both produce correct outputs. Note that both gray resources result in detectable errors because at least one module outputs a different value than the other modules. We consider the error correction to be 100% if all errors can be classified in this way as correctable errors, although failures that occur in other parts of the system may still cause incorrect outputs.

The input to the problem is a dataflow graph (DFG) D, a latency constraint L expressed in number of cycles, and an error constraint E specified as the minimum acceptable EC%. The output is a solution X, which is a combination of a schedule S and binding B for a redundant version of D. Given these inputs and outputs, we define the problem as follows:

$$\begin{aligned} \text{Minimize} \quad & \text{NumResources}\,(X) \\ \text{Subject to} \quad & \text{Latency}\,(X.S) \leq L, \\ & \text{ErrorCorrection}\,(X.B) \geq E, \\ & \text{ErrorDetection}\,(X.B) = 100\%. \end{aligned} \tag{1}$$

In other words, the goal of the problem is to find a schedule and binding that minimizes the number of required resources, where the schedule does not exceed the latency constraint L, the binding does not exceed the error constraint E, and all errors are detectable. We provide an informal proof that this problem is NP-hard as follows. If we remove both error constraints from the problem definition, the problem is equivalent to minimum-latency and resource-constrained scheduling followed by binding, which are both NP-hard problems [15]. The correctable and detectable error constraints only make the problem harder by expanding the solution space with replicated versions of the input.

Note that a more complete definition of this problem would include other FPGA resources (e.g., DSP units, block RAM), as opposed to solely using LUTs. However, because there is no effective relative cost metric for different FPGA resources, comparison between solutions with different types of FPGA resources is difficult. For example, it is not clear whether or not a solution with 100 DSPs and 10,000 LUTs is preferable to a solution with 10 DSPs and 100,000 LUTs. An alternative to this approach would be minimizing one resource while using constraints on the other resources. This approach however may exclude solutions that minimize multiple selected resources. For ease of explanation and comparison, the presented heuristic implements coarse-grained resources using only LUTs and focuses on minimizing the design's overall LUT count.

We assume that scrubbing occurs frequently enough so there cannot be more than one faulty resource at a time, which is often true due to the low frequency of SEUs in many contexts. For example, in the Cibola Flight Experiment [23], the experiment's Virtex FPGAs experienced an average SEU rate of 3.51 SEUs/day compared to their scrubbing cycle of 180 ms. With this assumption, based on our definitions, the total number of possible faults (and errors) is equal to the total number of resources used by the solution. Due to the likely use of SRAM-based FPGAs, we assume that all faults persist until scrubbing removes the fault. This contrasts with earlier work that focuses on *transient* faults [24–26]. We assume the presence of an implicit voter at the output of the modules, potentially using strategies from [14].

One potential challenge with error correction is the possibility of two modules producing incorrect errors that have the same value, which we refer to as *aliased errors*. Although we could extend the problem definition to require no instances of aliased errors, this extension is not a requirement for many usage cases (e.g., [26]). In addition, by treating aliased errors as uncorrectable errors, good solutions will naturally tend to favor bindings that have few aliased errors. To further minimize aliased errors, our presented heuristic favors solutions with the highest EC% when there are multiple solutions that meet the error constraint with equivalent resources.

4. Force-Directed Fault-Tolerance-Aware (FD-FTA) Heuristic

To solve the problem of minimum-resource, latency- and error-constrained scheduling and binding, we introduce the Force-Directed Fault-Tolerance-Aware (FD-FTA) heuristic, which performs scheduling and binding during high-level synthesis while simultaneously applying TMR and resource sharing to reduce overhead. By using an error-correction constraint combined with a latency constraint, the heuristic explores various tradeoffs between area, performance, and redundancy. As described in Algorithm 1, the heuristic first triplicates the DFG, schedules the triplicated DFG under a given latency constraint, and then binds the scheduled operations under a given EC% constraint.

The heuristic is divided into two key parts: scheduling and binding. We discuss the scheduling algorithm in Section 4.1 and the binding algorithm in Section 4.2.

4.1. Scheduling. In high-level synthesis, scheduling is the process of assigning each operation into a specific cycle or control state. The resulting schedule for the entire application is then implemented using a finite-state machine. In this section, we discuss the limitations of previous fault-tolerance-aware schedulers (Section 4.1.1) and then present a heuristic that adapts Force-Directed Scheduling (FDS) [27] to address those limitations (Section 4.1.2).

4.1.1. Previous Fault-Tolerance Aware Scheduling. The previous work on fault-tolerance-aware scheduling from [5] used the *Random Nonzero-Slack List Scheduling*. As a variant form of minimum-resource, latency constraint (MR-LC) list scheduling, this scheduling algorithm is a greedy algorithm that makes scheduling decisions on a cycle-by-cycle basis based on a resource bound and operation slack (i.e., the difference between the latest possible cycle start time and the cycle under consideration). To minimize resource usage, the algorithm iterates over each cycle and schedules operations ordered from lowest to highest slack up to the resource bound. If there are still zero-slack operations in a particular cycle once the resource bound is reached, those operations are scheduled and the resource bound is updated to match this increased resource usage. This process continues until all of the operations are scheduled. Unlike MR-LC list scheduling, the Random Nonzero-Slack List Scheduling schedules nonzero-slack operators with a 50% probability up to the resource bound. With this randomness, the scheduling algorithm is intended to produce different schedules for each TMR module which would increase the likelihood of intermodule operations bindings following scheduling. To escape local optima, this previous work used the scheduling

Input: Dataflow graph D, latency constraint L, error correction constraint E
Output: Solution X that minimizes resources
begin
Step 1. Triplicate D into D_{FT};
Step 2. Schedule the operators of D_{FT} with constraint L to obtain S;
Step 3. Bind the operators of S with constraint E to obtain B;
Step 4. Return solution X from S and B;
end

Algorithm 1: Force-Directed Fault-Tolerance-Aware (FD-FTA) heuristic.

Input: Dataflow graph representation of the design
Output: Operator assignments to cycles
while *there are unscheduled operations* **do**
Step 1. Evaluate time frames;
Step 2. Update distribution graphs;
Step 3. Calculate self-forces for every feasible cycle;
Step 4. Calculate total force from self-force, predecessor force, and successor force;
Step 5. Schedule operation with lowest force;
end

Algorithm 2: Force-Directed Scheduling.

algorithm in a multipass approach that would collect the best result from a series or phase of scheduling and binding runs. The heuristic would then continue to run these phases until the best result of a phase showed no significant improvement compared to the previous phase's result.

There are several key disadvantages of using the previous heuristic with this scheduling algorithm that we address in this paper. One primary disadvantage is the heuristic's lengthy execution time from its multipass approach, which relies on randomness in the schedule to find an improved solution. Using a user-defined percentage, the heuristic continues to another phase with double the amount of scheduling and binding runs if the current phase's result is not significantly better than the previous phases. The execution time can therefore be largely influenced by the randomness and the data dependencies between operations. In addition to long execution times, the heuristic requires fine-tuning of starting parameters to avoid premature exiting, which worsens productivity and can be error prone. The heuristic also has the disadvantage of exploring a restricted solution space by favoring the scheduling of operations in earlier cycles. This scheduling tendency results from the use of a fixed 50% scheduling probability for nonzero-slack operations in each cycle. By contrast, our proposed heuristic performs the scheduling and binding process *once*, using a more complex force-directed scheduler, which generally reduces the execution time and improves quality without the need for manually tuning heuristic parameters.

In terms of complexity, both heuristics consist of scheduling followed by binding. As such, the proposed heuristic consists of $\mathcal{O}(cn^2)$ complexity for the force-directed scheduler [27] described in the next subsection and $\mathcal{O}(n^2)$ for the binder described in Section 4.2 [28]. The overall complexity is therefore $\mathcal{O}(cn^2)$ where c is the latency constraint and n is the number of operations. In contrast, the previous heuristic consists of $\mathcal{O}(n)$ complexity for its variant list-scheduler and $\mathcal{O}(n^2)$ for its clique partitioning binder. Since the previous heuristic will generally limit the number of phases or total number of scheduling and binding runs in its multipass approach, the overall complexity is $\mathcal{O}(pn^2)$, where p is a user-defined limit on total iterations and n is the number of operations. The proposed heuristic therefore has a lower complexity than the previous heuristic when $c < p$ which is commonly true as the user will generally set p such that $p \gg c$ to avoid premature exiting.

Additionally, the previous heuristic suffers from the same disadvantages as general MR-LC list scheduling. Notably, it is a local scheduler that makes decisions on a cycle-by-cycle basis using operation slack as a priority function. Such an approach has a tendency towards locally optimal operation assignments that often leads to suboptimal solutions when providing a latency constraint that is significantly larger than the minimum-possible latency. Similarly, the heuristic's use of slack may present suboptimal results in scenarios where operation mobility may underestimate resource utilization. By contrast, Force-Directed Scheduling is a global stepwise-refinement algorithm that selects operation assignments from any cycle based on its impact on operation concurrency.

4.1.2. Force-Directed Scheduling. Force-Directed Scheduling is a latency-constrained scheduling algorithm that focuses on reducing the number of functional units by balancing the concurrency of the operations assigned to the units. Algorithm 2 presents an overview of the algorithm. A more detailed description can be found in [27].

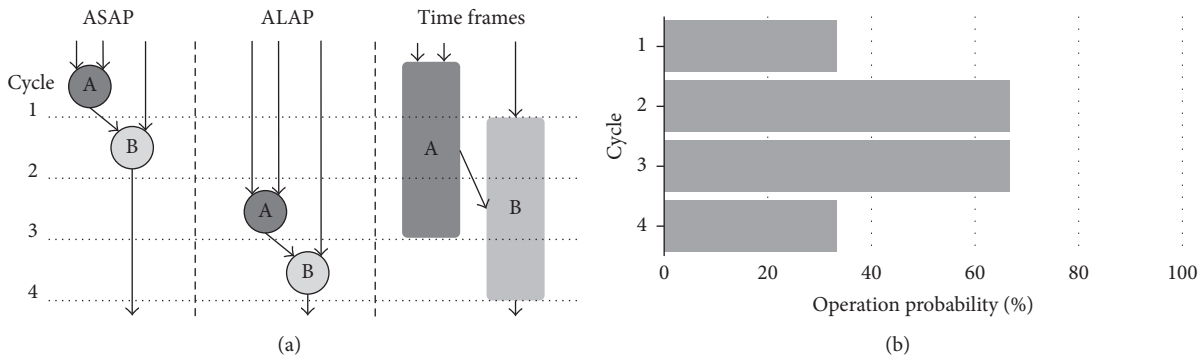

Figure 2: Illustrations of different structures used in Force-Directed Scheduling. The diagram in (a) shows the ASAP and ALAP schedule of a DFG and the respective time frames of the operations. The chart in (b) shows the distribution graph assuming both operations use the same resource type.

As demonstrated in Algorithm 2, Force-Directed Scheduling balances operation concurrency by using time frames, distribution graphs, and force values. *Time frames* refer to the possible cycles to which an operation may be assigned, such that the resulting schedule does not violate the latency constraint. Intuitively, the assignment of an operation to a specific cycle may impact the time frames of other operations if there are data dependencies. *Distribution graphs* refer to the probability that a given operation type is assigned to a specific cycle for each cycle within the latency constraint. For each cycle, the algorithm assigns probabilities to the distribution graphs by finding all operations of the same type that can be scheduled at a given cycle and then sums their individual probabilities.

To better illustrate these structures, Figure 2 shows the time frames and distribution graphs of a DFG that consists of two operations and is subject to a latency constraint of 4 cycles. In Figure 2(a), the DFG is shown scheduled with an as-soon-as-possible (ASAP) schedule and an as-late-as-possible (ALAP) schedule. The ASAP schedule focuses on scheduling an operation at the earliest possible cycle given data dependencies. Notice how operation B cannot be scheduled on cycle 1 because it is dependent on operation A. Similarly, the ALAP schedule focuses on scheduling an operation at the latest possible cycle. Force-Directed Scheduling uses these two schedules to form the time frame of each operation, which represents the cycle bounds of an operation assignment.

Figure 2(b) shows the distribution graphs of the DFG before any operation has been scheduled. Assuming operations 1 and 2 are of the same type (or can share a resource), each operation has a uniform 33% chance of being scheduled in any cycle of its 3-cycle time frame. The distribution graph therefore shows a 33% probability of that operation type being scheduled in cycles 1 and 4 and a 66% probability for the cycles where the time frames of the operations overlap.

Force-Directed Scheduling abstracts each distribution graph as a series of springs (for each cycle) connected to each operation of the DFG. Therefore, any spring displacement exerts a "force" on each operation. In this abstraction, the spring's strength is the distribution graph's value in a particular cycle and the displacement is a change in probability in a cycle due to a tentative assignment. A tentative operation assignment will therefore cause displacements in the cycle it is assigned to, whose probability is now 100%, and in the cycles it can no longer be assigned to, whose probabilities are now 0%. Using this abstraction, each operation assignment has an associated total *force* value that reflects the assignment's impact on the time frames and distribution graphs of the operation and its preceding and succeeding operations. While there are unscheduled operations, the algorithm schedules the operation assignment with the lowest force value reflecting the lowest impact on operation concurrency.

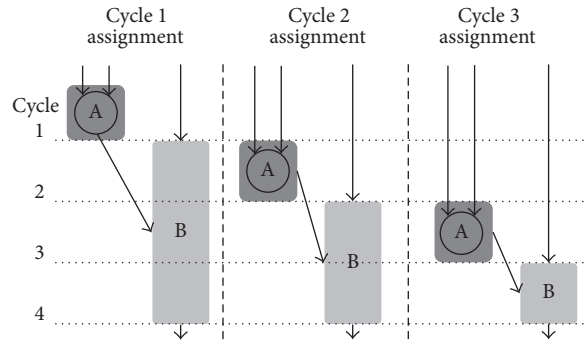

Figure 3: Illustrations of the DFG's time frames after a tentative assignment of operation A.

Figure 3 depicts several examples of how an operation assignment may impact the time frames of other operations using the DFG from Figure 2. In each of these examples, operation A is scheduled in a specific cycle, which limits the operation's time frame to that cycle. For the first diagram, operation A is scheduled in cycle 1 and has no impact on operation B's time frame. In contrast, scheduling operation A in cycle 2 or 3 reduces operation B's time frame, since operation A must be scheduled in a cycle before operation B.

Figure 4 depicts the corresponding distribution graph after such cycle assignments. Notice that an operation assignment in a particular cycle changes the probability of that operation type being scheduled to 100%. In (a), operation A contributes a 100% probability in cycle 1, while operation B

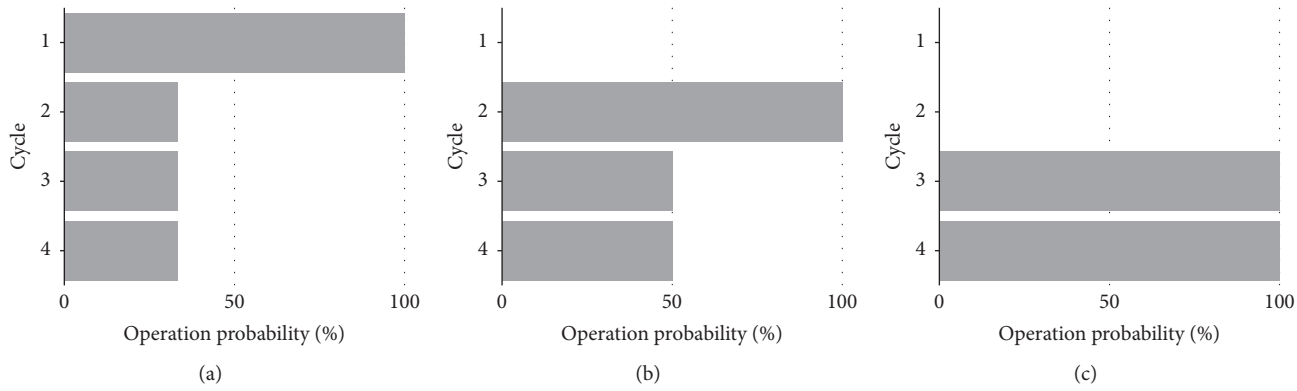

Figure 4: Distribution graphs for a cycle 1 assignment (a), for a cycle 2 assignment (b), and for a cycle 3 assignment (c) of operation A.

contributes a uniform 33% probability over its time frame from cycle 2 to cycle 4. In (b), operation A still contributes a 100% probability in its scheduled cycle, while operation B contributes a uniform 50% probability over its reduced time frame. Noticeably, in (c), operation A's assignment in cycle 3 will effectively force operation B's later assignment to cycle 4 due to data dependencies. It should be noted that a distribution graph may have probabilities over 100% which can represent multiple operation assignments of that type in the same cycle. The main goal of Force-Directed Scheduling is to balance operation concurrency such that the distribution graph for each operation type will have a relatively uniform probability over each cycle after all operations are scheduled. Using the equations for *force*, the assignment of operation A to cycle 1 would have the lowest force value of the three possible assignments since it balances the operation probability across the most cycles.

In [27], the authors describe an adjustment for optimization in scenarios with resources with different realization costs. This adjustment involves scaling the force values by a cost factor reflecting the relative realization costs. In our proposed heuristic, we scale the force values by the LUT requirements of the resource.

4.2. Binding. In high-level synthesis, binding refers to the process of assigning operations and memory access to hardware resources. When following scheduling, the binding process yields a mapping of operations to hardware resources at specific cycles.

4.2.1. Singleton-Share Binding. Our proposed heuristic, referred to as *Singleton-Share Binding*, involves a two-stage process that first performs binding in each TMR module separately and then merges nonconflicting bindings between TMR modules. By binding operations from different TMR modules to the same resource, the heuristic has therefore made tradeoffs between the circuit's capability to correct an error and the total area needed by the circuit. Using the aforementioned EC% constraint, a designer can limit the total amount of intermodule binding that can occur.

Algorithm 3 displays the pseudocode of Singleton-Share Binding.

In the first stage, Singleton-Share Binding considers each TMR module separately and binds the module's operations to resources using the Left Edge Algorithm (LEA) as originally described in [29]. Although originally introduced as a method for packing wire segments, this algorithm has found popularity in resource binding that focuses on binding as many operations to a resource as possible before binding on another resource.

In the second stage, the binding algorithm consolidates singleton bindings (i.e., resources with only one mapped operation) and attempts to merge each singleton binding with a nonconflicting binding from another module. By merging these bindings, a single resource will contain operation bindings from different TMR modules which will reduce the circuit's error-correcting capability at the benefit of a lower area cost. This binding process will therefore reduce the total number of resources used in the design at the cost of reducing the design's EC%. As mentioned before, the heuristic maintains 100% error detection by limiting the resource sharing to only two modules. This process allows the design to detect errors, albeit uncorrectable, in the event of a fault in a shared resource. Algorithm 3 shows the pseudocode for the second stage which first merges bindings between singleton and nonsingleton bindings and then merges bindings between singleton binding pairs. Bindings may only be merged if the resulting binding does not involve multiple operations scheduled on the same cycle or operations of all three modules.

The key requirement of the second stage is ensuring that the EC% does not fall under the error constraint E. Since the number of uncorrectable errors equals the number of resources that have been shared across two modules, this constraint satisfaction is illustrated as shown in

$$\frac{\text{resources}_{\text{used}} - \text{resources}_{\text{Shared}}}{\text{resources}_{\text{used}}} \times 100\% \geq E \qquad (2)$$

which can be reduced to

$$\text{resources}_{\text{Shared}} \leq \text{resources}_{\text{used}} \times \left(1 - \frac{E}{100}\right). \qquad (3)$$

These equations however do not reveal a limit on the number of shares that can be done by the second stage binder. Since sharing a resource also reduces the number of

```
Input: Operator assignments to control-steps, S
Output: Operator bindings to resources, B_stg2
begin
  /* Stage 1 Binding                                              */
  B_stg1 ← ∅;
  foreach module m do
    B_stg1 ← B_stg1 ∪ LeftEdgeAlgorithm(S_m)
  end
  /* Stage 2 Binding                                              */
  shares ← 0;
  limit ← GetLimit(B_stg1);
  singleton, non-singleton ← SeparateRes(B_stg1);
  /* Stage 2: Merge singletons with non-singletons                */
  foreach resource s of singleton do
    foreach resource n of non-singleton do
      if shares = limit then break;
      ;
      if IsMergeable(s, n) then
        add operator of s to n and remove s from singleton;
        shares ← shares + 1;
      end
    end
  end
  /* Stage 2: Merge singletons with other singletons              */
  foreach pair of singleton (s, n) do
    if shares = limit then break;
    ;
    if IsMergeable(s, n) then
      add operator of s to n and remove s and n from singleton;
      add n to non-singleton;
      shares ← shares + 1;
    end
  end
  B_stg2 ← singleton ∪ non-singleton;
  return B_stg2
end
```

Algorithm 3: Singleton-Share Binding.

resources, the relationship between the number of shares i and the initial number of bindings is related in

$$i \leq \left(\text{usage}_{\text{stg1}} - i\right) \times \left(1 - \frac{E}{100}\right). \tag{4}$$

Since the number of shares must be an integer, we define the integer *limit* as the maximum number of shares done by the second stage binder while meeting the error constraint. We simplify (4) in terms of *limit* in

$$\text{limit} \leq \left\lfloor \text{usage}_{\text{stg1}} \times \frac{100 - E}{200 - E} \right\rfloor. \tag{5}$$

Figure 5 depicts an example of the binding process. In these subfigures, each nonsingleton resource is represented by a node with a solid color, whereas each singleton resource is represented by a node with a pattern. The format A_n on each node refers to an operation A from the original DFG performed on module n. Figure 5(a) therefore displays three modules each with distinct singleton and nonsingleton resources and represents a possible scheduling and binding after the first stage of binding.

Using a 50% EC constraint and (5), the second stage of binding may share up to two resources to meet the EC% constraint. As the first step, the binding process attempts to merge singleton bindings with nonsingleton bindings. In Figure 5(a), the only candidates for sharing is singleton bindings of either node B_1 or node A_3, with the nonsingleton binding containing nodes A_2, C_2, and D_2 (the light-gray resource). The singleton binding containing B_2 is ineligible since all nonsingleton bindings already have a conflicting operation scheduled during cycle 2. Similarly, no other nonsingleton binding can be merged with a singleton as each contains a conflicting operation during cycles 1 and 2. Merging singleton binding of node B_1 with the candidate nonsingleton binding, the binding process produces Figure 5(b). Notice that node B_1 is now part of the nonsingleton binding of A_2, C_2, and D_2, as represented by the same color.

Since there are no more candidate pairs, the binding process then attempts to merge singleton binding pairs. In

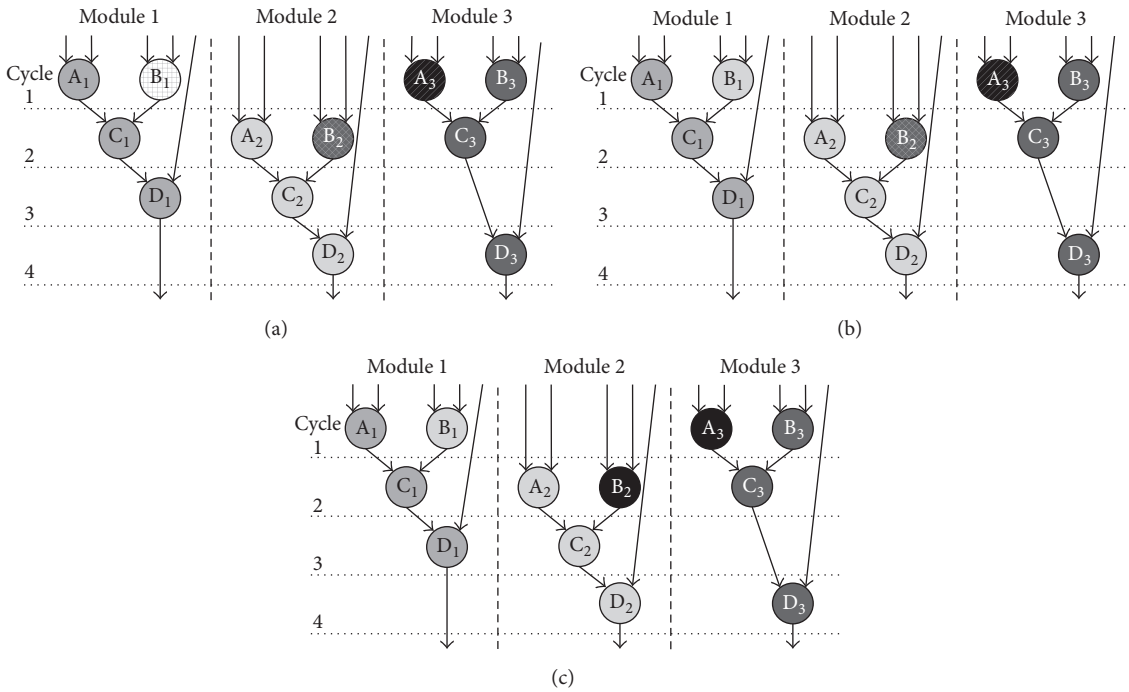

Figure 5: Sample binding for a latency constraint of 4 cycles and error constraint of 50%. The binding in (a) depicts the first stage of binding, whereas the bindings in (b) and (c) represent bindings following different steps in the second stage. The binding in (b) depicts the merging of a singleton with a nonsingleton, and the binding in (c) depicts the merging of a pair of singletons.

Figure 5(b), there is only one candidate pair between the singleton of B_2 and of A_3. Merging these bindings, the binding process produces a nonsingleton binding containing these two nodes, as seen in Figure 5(c) with the solid black color. At this point, the binding process completes as the number of shared resources has met the limit. Instead of using the initial size resources, the final design now produces a scheduling and binding on four resources with a 50% EC. If the limit had not been reached, the binding process may continue until there are no more candidate pairs.

For register-binding, our heuristic provides a register unit for each functional unit to avoid relatively expensive multiplexers on the FPGA's LUT-based architecture while also providing register sharing for compatible variables originating from the same functional unit. Due to the FPGA's register-rich fabric, register sharing can be more expensive than register duplication and is rarely justified [30].

5. Experiments

In this section, we evaluate the effectiveness of the FD-FTA heuristic in a variety of benchmarks. Section 5.1 describes the experimental setup. Section 5.2 shows a comparison in resource savings with TMR applied to existing RTL. Section 5.3 shows a comparison in resource savings with a previous approach. Section 5.4 shows a comparison in execution time with the previous approach. Section 5.5 illustrates a comparison of clock frequencies using the different approaches.

5.1. Experimental Setup. To evaluate our heuristic, we implemented Algorithm 1 in C++, while also using Vivado 2015.2 to provide LUT counts for each resource and the clock frequency of the final solution. Our target architecture was a Xilinx Virtex-7 xc7vx485tffg1761-2, which uses Xilinx 7 Series Configurable Logic Blocks with 6-input LUTs.

5.1.1. Benchmarks. Table 1 summarizes the benchmarks and shows the numbers of nodes and edges and the operation types. To represent signal-processing applications, we used DFGs for 5×5 convolution, 8-point radix-2 butterfly FFT, 16-point radix-2 butterfly FFT, and 4-point radix-4 dragonfly FFT. Of these, the radix-4 FFT efficiently expands the complex arithmetic involved to equivalent real operations as in [31]. For the radix-2 FFTs, we use resources capable of directly performing complex operations. To represent fluid-dynamics and similar applications, we used two DFGs that solve 5-dimensional linear equations ($Ax = B$) using the Jacobi iterative method and the successive overrelaxation (SOR) iterative method [32]. We also used two DFGs that solve Laplace's equation ($\nabla^2 f = 0$) using the Jacobi and SOR methods. We also supplement these real benchmarks with synthetic benchmarks (small0–small7, medium0–medium4) that we created using DFGs generated through random directed acyclic graphs.

5.1.2. Baselines. Two baselines were used to evaluate our proposed heuristic: a common approach where designers apply TMR to existing RTL (collectively referred to as *TMR-RTL*

Table 1: Benchmark summary.

Benchmark	Description	Number of nodes	Number of edges	Operation types
conv5x5	5x5 convolution kernel.	49	190	Add, Mult
conv9x9	9x9 convolution kernel.	161	932	Add, Mult
fft8	8-point radix-2 DIT FFT based on butterfly architecture.	32	116	Add, Sub, Mult
fft16	16-point radix-2 DIT FFT based on butterfly architecture.	88	648	Add, Sub, Mult
fftrad4	4-point radix-4 FFT based on dragonfly architecture. Efficiently decomposing complex operations into real operations.	40	232	Add, Sub, Mult
linsor	Single iteration kernel to solve a system of linear equations ($Ax = B$) in 5 variables, using successive-overrelaxation (SOR) method.	64	2657	Add, Sub, Mult, Div
linjacobi	Single iteration kernel to solve a system of linear equations in 5 variables, using Jacobi method.	60	230	Add, Sub, Mult, Div
lapsor	Single iteration kernel to solve Laplace's equation ($\nabla^2 f = 0$), using successive-overrelaxation (SOR) method.	7	16	Add, Sub, Mult
lapjacobi	Single iteration kernel to solve Laplace's equation using Jacobi method.	5	9	Add, Sub, Mult
small0–7	8 randomly generated DFGs with <20 operations.	4–19	2–36	Add, Mult
medium0–4	5 randomly generated DFGs with <130 operations.	88–127	346–642	Add, Sub, Mult, Div

for simplicity) and the approach from [5] (referred to as the *previous* approach). We use the TMR-RTL approach baseline to motivate the benefits of our heuristic over common use cases and the previous approach baseline to demonstrate improvements over the state of the art.

We model the TMR-RTL approach based on two observations: (1) designers often deal with existing RTL code and (2) designers may not be willing or capable of performing an extensive exploration of fault-tolerance tradeoffs. Because most existing RTL is implemented without considering the effects of TMR, the code is generally not written to minimize area. To approximate this use case, we use an ASAP schedule and LEA bindings. Although a designer could certainly use an approach that reduces area, we have observed that ASAP schedules are common in RTL cores, likely due to ease of implementation.

5.2. Comparison with TMR Applied to Existing RTL. In this section, we evaluate the effectiveness of our FD-FTA heuristic compared to TMR-RTL under different EC% and latency constraints. Unlike a simple triplicated RTL circuit, an HLS RTL circuit may use more extensive resource sharing during high-level synthesis to automatically target different latency and redundancy constraints. Applying triplication to an existing circuit, in contrast, limits the designer to the circuit's existing architecture and leaves little opportunities to target different constraints, especially when using encrypted or presynthesized IP cores. Table 2 presents the resources savings where the rows correspond to different benchmarks and the columns correspond to different EC% and latency constraints. We vary the latency constraint from the DFG's minimum-possible latency (as determined by an ASAP schedule) up to 2x the minimum latency, in increments of 0.2x. The EC% constraint is explored for 100% and 70%. On average, the FD-FTA heuristic shows savings compared to the TMR-RTL for each pair of normalized latency and EC% constraints. Although relatively small at the lowest latency constraint, these average savings become significant at higher latency constraints reaching 45% and 49% savings at 2.0x normalized latency constraint for both 100% and 70% EC constraints, respectively. For individual benchmarks, the FD-FTA heuristic shows significant savings at each set of constraints except for benchmarks with low operation counts and for certain benchmarks at low normalized latency constraints.

From Table 2, we observe results consistent with expectations. For a normalized latency constraint of 1.0x and a EC constraint of 100%, the FD-FTA heuristic yielded no savings for about half of the benchmarks and up to 65% savings for the remainder. The lack of savings for certain benchmarks is expected given that some benchmarks have no flexibility to schedule operations on different cycles when constrained by the minimum-possible latency. In such cases, designers should use the lowest complexity scheduling algorithm as all algorithms will produce similar schedules. For the other benchmarks, up to 65% savings correspond to benchmarks with little to some flexibility, whereas the 65% savings in the *linsor* benchmark correspond to DFGs with large operation flexibility. In these cases, we start to see a trend of growing resource savings as operation flexibility increases. Additionally, the FD-FTA heuristic displayed a minimal degradation between −2% and −1%, on some benchmarks. Further investigation of each approach's output reveals that nonoptimal bindings by the FD-FTA heuristic resulted in multiplexers with more inputs than their counterparts in the TMR-RTL approach.

For a normalized latency constraint of 2.0x and a EC constraint of 100%, the FD-FTA heuristic yielded savings up to 75% with savings above 50% for about half of the benchmark set. We expect such high savings due to the large

Table 2: Resource savings (LUT%) of FD-FTA heuristic compared to TMR-RTL for 100% and 70% EC.

Benchmark	Normalized latency and EC% constraints											
	1.0x		1.2x		1.4x		1.6x		1.8x		2.0x	
	100%	70%	100%	70%	100%	70%	100%	70%	100%	70%	100%	70%
lapjacobi	0%	0%	0%	0%	5%	30%	2%	5%	2%	2%	5%	4%
lapsor	0%	0%	−1%	−1%	−1%	−1%	−1%	2%	−2%	0%	−2%	0%
conv5x5	18%	18%	52%	52%	58%	65%	64%	67%	62%	71%	68%	71%
fft8	0%	0%	12%	13%	13%	29%	19%	25%	24%	32%	38%	37%
fftRadix4	0%	0%	−1%	15%	21%	29%	21%	39%	28%	43%	32%	43%
fft16	0%	0%	16%	28%	9%	18%	11%	25%	−10%	5%	18%	27%
linjacobi	27%	28%	50%	52%	47%	59%	61%	67%	63%	68%	62%	70%
linsor	65%	69%	68%	68%	70%	72%	70%	72%	71%	72%	70%	74%
conv9x9	19%	19%	50%	50%	66%	70%	69%	74%	76%	78%	75%	80%
Average	14%	15%	27%	31%	32%	41%	35%	42%	35%	41%	40%	45%
small0	1%	1%	1%	1%	10%	13%	10%	13%	2%	11%	48%	57%
small1	0%	1%	0%	1%	0%	1%	3%	30%	3%	30%	2%	4%
small2	0%	0%	0%	0%	20%	22%	20%	22%	19%	30%	39%	40%
small3	30%	30%	30%	30%	57%	57%	53%	57%	45%	58%	58%	57%
small4	30%	49%	30%	49%	38%	38%	49%	57%	40%	49%	57%	57%
small5	27%	44%	35%	54%	37%	45%	28%	30%	38%	39%	45%	46%
small6	5%	5%	5%	5%	11%	20%	29%	38%	29%	28%	31%	49%
small7	0%	0%	0%	0%	31%	35%	39%	47%	50%	57%	54%	58%
medium0	3%	3%	37%	42%	50%	52%	50%	55%	57%	61%	52%	60%
medium1	3%	3%	32%	44%	49%	54%	51%	55%	58%	62%	60%	64%
medium2	0%	0%	38%	42%	51%	54%	58%	63%	60%	63%	62%	67%
medium3	6%	6%	33%	39%	45%	50%	49%	55%	52%	57%	52%	59%
medium4	0%	0%	33%	37%	43%	51%	49%	51%	52%	57%	57%	60%
Average	8%	11%	21%	27%	34%	38%	37%	44%	39%	46%	48%	52%
Total average	11%	13%	24%	28%	33%	39%	37%	43%	37%	44%	45%	49%

operation flexibility granted by the latency constraint which enables the FD-FTA heuristic's global scheduling approach to balance the operation concurrency over 2x as many cycles as the minimum-possible latency. The TMR-RTL approach, in contrast, will maintain a static scheduling approach despite the larger latency constraint. Like any basic triplication approaches, the TMR-RTL approach is unable to target new constraints without manual intervention and may suffer from a higher area cost due to fewer opportunities for resource sharing. Despite these savings, there are a few benchmarks where the FD-FTA heuristic experienced little to no savings. These benchmarks (e.g., *labjacobi*, *lapsor*, and *small1*) are however very small in operation count and represent cases where the FD-FTA heuristic's attempt to balance operation concurrency will yield output similar to the TMR-RTL approach. Overall, these results have supported our expectation that the FD-FTA heuristic performs better with increasing latency constraints due to larger scheduling flexibility.

Comparing the results from nonsynthetic benchmarks and synthetic benchmarks, Table 2 shows that the trends found in each individual benchmark set are similar to the trends found when both sets are considered together. For nonsynthetic benchmarks, the FD-FTA heuristic yielded no savings for about half of the benchmarks and up to 65% savings for the remainder at a latency constraint of 1.0x and EC constraint of 100%. As the latency constraint increased, the FD-FTA heuristic generally showed increased savings with exception to *lapsor* and *fftRadix*4 which displayed minimal degradation between −2% and −1% under certain latency constraints. At a latency constraint of 2.0x, the FD-FTA heuristic yielded savings up to 75%. Similarly, in the synthetic benchmarks, the FD-FTA heuristic also yielded no savings for about half of the benchmarks and up to 30% savings for the remainder at a latency constraint of 1.0x and EC constraint of 100%. As the latency constraint increased, the FD-FTA heuristic also generally showed increased savings but at a slower rate than the nonsynthetic benchmarks. At a latency constraint of 2.0x, the FD-FTA heuristic showed savings up to 62%. With similar trends, both benchmark sets also showed averages within 5% of their respective total averages.

To better illustrate these trends, Figure 6 shows the LUT savings of certain benchmarks that exemplify the different behaviors observed. In this graph, four benchmarks are displayed with EC constraints of 70% and 100%. Each benchmark entry is shown with six bars representing the LUT savings compared to the TMR-RTL approach for the six normalized latency constraints.

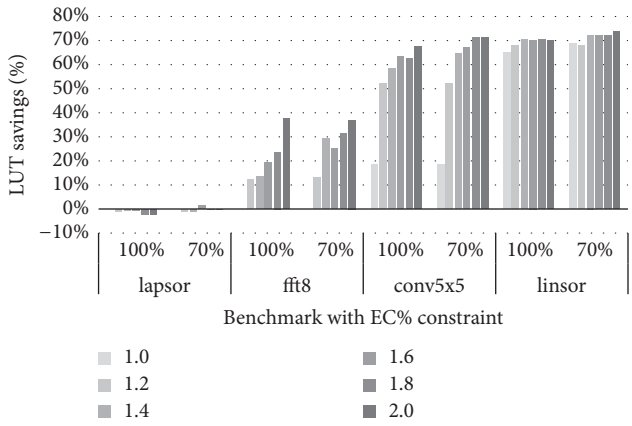

Figure 6: Resource savings of FD-FTA heuristic under different normalized latency constraints for selected benchmarks and EC% constraints when compared to TMR-RTL.

As one may expect, the impact of a latency constraint for HLS has a quite varied impact depending on the DFG structure. For the *lapsor* and other small benchmarks, increasing the latency constraint generally had a minimal impact in resource savings since the FD-FTA heuristic is likely to produce a similar schedule as TMR applied to existing RTL at small DFG operation counts. Similarly, the *linsor* benchmark also experienced small increases in savings as the latency constraint increased, despite the large initial savings. This behavior is caused by the benchmark's innate structure that enables much operation scheduling flexibility, even for the minimum-possible latency constraint. Due to the large flexibility, the FD-FTA heuristic's scheduling algorithm can balance the operation concurrency easier and experiences diminished benefits from additional flexibility granted by an increased latency constraint. In contrast, the *fft*8 and *conv*5*x*5 benchmarks show significant increases in savings when the latency constraint is increased. We attribute this behavior to minimal scheduling flexibility at the minimum-possible latency constraint which is alleviated with larger latency constraints.

When comparing the savings of 70% EC with 100% EC, Table 2 shows that the experiments under the 70% EC constraint had additional savings up to 27% compared to experiments under the 100% EC constraint with the same latency constraint. Similar to previous observations, the decreased EC% largely had no impact in experiments with a normalized latency constraint of 1x which is primarily due to the lack of operation flexibility. For this baseline latency constraint, the singletons of each module are likely scheduled on the same cycle and are therefore not eligible for singleton-sharing as seen in a majority of the benchmarks. Otherwise, the 70% EC iterations generally showed additional savings compared to their counterpart 100% EC iterations as the latency constraint is increased. Notably, these additional savings remain relatively similar once savings occur. Such savings are expected as EC% reflects the number of resources that may be shared across modules which places an upper bound on the additional savings due to EC.

Despite the overall positive impact by decreasing the EC%, there are a few cases where decreasing the EC% also decreased the LUT savings. For those cases, the savings due to sharing were offset by the cost of increasing the number of inputs on the interconnection multiplexers for the remaining resources. This trend is however only largely noticeable in the smaller DFGs, where the cost of input multiplexers may be comparable to the cost of resources. Similarly, there are also cases were decreasing the EC% showed a sudden increase in resource savings for a single latency constraint, but not for others, as seen in the *lapjacobi* benchmark. In these scenarios, the specific latency constraint of increased resource savings may present the only scenario where the Singleton-Share Binding is capable of sharing singletons. In contrast, the heuristic may present incompatible singleton bindings due to inflexible schedules at lower latency constraints or suboptimal schedules at higher latency constraints due to the force-directed global scheduling approach. Overall, determining the impact of the EC% constraint is counterintuitive as the constraint's impact relies primarily on the heuristic's distribution of singletons among cycles within each module. This distribution cannot be easily determined from the benchmark or constraints as the scheduling and binding process may vary the singleton distribution at the slightest change of constraints or in DFG structure. This impact is further affected by the shared resource type and by whether the sharing has increased the size of the resource's input multiplexer.

5.3. Comparison with Previous Approach. In this section, we evaluate the effectiveness of our FD-FTA heuristic compared to the approach of previous work described in [5] under different EC% and latency constraints. Table 3 presents the resource savings of the FD-FTA heuristic compared to the previous approach and uses the same presentation style as Table 2. On average, the FD-FTA heuristic shows savings compared to the previous work for each pair of normalized latency and EC% constraints. Although minimal at the lowest latency constraint, these savings become significant at higher latency constraints reaching 21% and 23% average savings at 2.0x normalized latency constraint for 100% and 70% EC, respectively. For individual benchmarks, the FD-FTA heuristic commonly shows significant savings at each set of constraints except for benchmarks with low operation counts and for most benchmarks at low normalized latency constraints.

Like the previous experiment, Table 3 expresses some similar results for benchmarks under the minimum-possible latency constraint. For the normalized latency constraint of 1.0x and 100% EC constraint, the FD-FTA heuristic generally has a small degradation ranging from −1% down to −5% in *fft*8. Similar to the results in Table 2, the FD-FTA heuristic has a tendency to make nonoptimal bindings that result in larger input multiplexers than the previous approach. Although present in all iterations of the experiment, such behavior is usually amortized by better utilization on fewer resources, especially on larger latency constraints. For the few cases where the FD-FTA heuristic experienced savings under these constraints, these iterations involved benchmarks with some flexibility to schedule operations on different cycles. In the case of the largest savings in the *small*5 benchmark,

Table 3: Resource savings (LUTS) of FD-FTA heuristic compared to previous work for 100% and 70% EC.

Benchmark	Normalized latency and EC% constraints											
	1.0x		1.2x		1.4x		1.6x		1.8x		2.0x	
	100%	70%	100%	70%	100%	70%	100%	70%	100%	70%	100%	70%
lapjacobi	0%	0%	0%	0%	2%	28%	−2%	−2%	−5%	−6%	−2%	−42%
lapsor	19%	19%	−1%	−1%	1%	−5%	1%	−2%	0%	−38%	0%	−33%
conv5x5	0%	0%	40%	41%	49%	56%	54%	59%	53%	64%	59%	63%
fft8	−5%	−5%	10%	9%	12%	27%	17%	22%	22%	29%	36%	35%
fftRadix4	0%	−2%	−6%	11%	12%	22%	9%	30%	12%	31%	13%	27%
fft16	−3%	−1%	15%	28%	9%	18%	11%	25%	−9%	6%	18%	27%
linjacobi	−1%	0%	29%	32%	24%	41%	43%	52%	44%	53%	42%	54%
linsor	0%	7%	−7%	−12%	−8%	−1%	−18%	−33%	−20%	−33%	−46%	−25%
conv9x9	−1%	0%	38%	38%	57%	63%	61%	67%	69%	72%	67%	74%
Average	1%	2%	13%	16%	18%	28%	20%	24%	18%	20%	21%	20%
small0	1%	1%	1%	1%	0%	4%	0%	4%	−21%	−10%	26%	40%
small1	0%	1%	0%	1%	0%	1%	2%	2%	2%	2%	0%	−35%
small2	0%	0%	0%	0%	11%	13%	11%	13%	−1%	12%	13%	15%
small3	0%	0%	0%	0%	34%	30%	25%	24%	1%	19%	20%	−4%
small4	10%	35%	10%	35%	11%	11%	15%	−5%	−18%	−25%	−4%	−4%
small5	27%	45%	28%	48%	22%	31%	−2%	−1%	3%	0%	−1%	−25%
small6	2%	2%	2%	2%	−1%	10%	11%	22%	1%	−1%	−11%	−19%
small7	0%	0%	0%	0%	28%	32%	34%	42%	43%	52%	46%	51%
medium0	0%	0%	33%	39%	44%	46%	44%	48%	49%	53%	40%	50%
medium1	0%	−2%	27%	40%	42%	48%	43%	47%	49%	53%	48%	53%
medium2	0%	0%	37%	41%	50%	53%	55%	60%	56%	60%	58%	63%
medium3	0%	0%	27%	33%	38%	43%	41%	48%	42%	49%	39%	49%
medium4	0%	0%	31%	35%	39%	48%	43%	47%	46%	52%	48%	52%
Average	3%	6%	15%	21%	24%	28%	25%	27%	19%	24%	25%	22%
Total average	2%	4%	14%	19%	22%	28%	23%	26%	19%	23%	23%	21%

the solution produced by the FD-FTA heuristic used far less multipliers than the previous approach's solution. This better utilization of the costly multipliers ultimately contributed to a smaller design.

Table 3 expresses some clear trends in the results of this experiment. These trends are general increases in resource savings as both the latency constraint and the number of operations in the benchmark increase. The experiment displays these trends in a majority of benchmarks and constraint sets with a few exceptions. These trends however are expected as the FD-FTA heuristic was designed to address disadvantages of the previous approach's scheduling algorithm. As mentioned in Section 4.1, the previous approach's scheduling algorithm had a number of scalability issues. The first trend of savings increasing with increased latency constraint can be attributed to the MR-LC scheduling algorithm that the previous approach is based on. Since MR-LC tries to minimize resources in a local cycle basis, this algorithm keeps a maximum operation count that limits the number of operations that may be scheduled in a cycle. These counts are only increased if the number of zero-slack operations in a cycle exceeds its current resource count. This behavior noticeably causes problems for latency constraints larger than the minimum-possible latency constraint since all operations are guaranteed to have nonzero slacks in the first few cycles. In this case, the algorithm will only schedule at most one operation per cycle until it experiences numerous zero-slack operations at later cycles. This lack of initial operation concurrency will likely cause large operation concurrency at later cycles. The design will therefore be forced to instantiate additional resources to meet this large concurrency. The result is a large number of resources that are underutilized in early cycles. By contrast, the FD-FTA heuristic's scheduling algorithm aims to balance the operation concurrency among all cycles such that all resources maintain a high utilization and therefore uses fewer overall resources.

Comparing the results from nonsynthetic benchmarks and synthetic benchmarks, Table 3 shows that the trends found in each individual benchmark set are similar to the trends found when both sets are considered together. For nonsynthetic benchmarks, the FD-FTA heuristic yielded no savings for most benchmarks and up to 19% savings for the remainder at a latency constraint of 1.0x and EC constraint of 100%. As the latency constraint increased, the FD-FTA

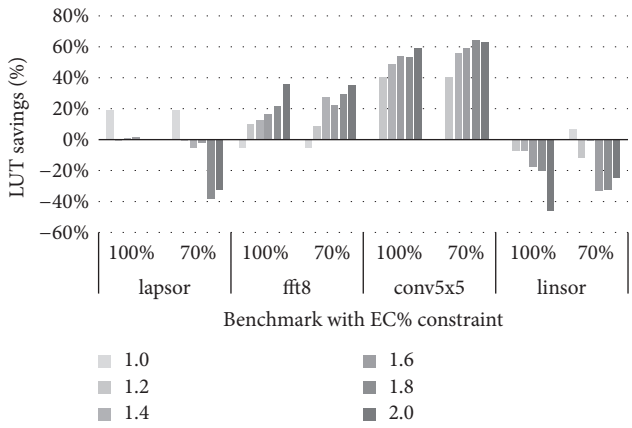

Figure 7: Resource savings of FD-FTA heuristic under different normalized latency constraints for selected benchmarks and EC% constraints when compared to the previous approach.

heuristic generally showed increased savings with exception to *lapjacobi* and *linsor* which decreased in savings. At a latency constraint of 2.0x, the FD-FTA heuristic yielded savings up to 67%. Similarly, in the synthetic benchmarks, the FD-FTA heuristic also yielded no savings for most benchmarks and up to 27% savings for the remainder at a latency constraint of 1.0x and EC constraint of 100%. As the latency constraint increased, the FD-FTA heuristic also generally showed increased savings but at a slower rate than the nonsynthetic benchmarks. At a latency constraint of 2.0x, the FD-FTA heuristic showed savings up to 58%. With similar trends, both benchmark sets also showed averages within 3% of their respective total averages.

To better illustrate these trends, Figure 7 shows the resource savings of certain benchmarks that exemplify the different behaviors observed. In this graph, four benchmarks are displayed with EC constraints of 70% and 100%. Each benchmark entry is shown with six bars representing the LUT savings compared to the previous approach for different normalized latency constraints.

Similar to Figure 6, the impact of a latency constraint for HLS has a quite varied impact depending on the DFG structure and on the effectiveness of the previous approach. For the *lapsor* and other small benchmarks, increasing the latency constraint generally had a small impact in resource savings since the FD-FTA heuristic is likely to produce a similar schedule to the previous approach. This behavior is mainly caused by the small operation count which limits the number of possible schedules. For a 70% EC constraint and 2.0x normalized latency constraint, the FD-FTA heuristic even displays a notable loss in savings. It should be cautioned that, at such small DFGs, any "significant" gain or loss of savings may be only the difference of one additional adder.

For the fft8 and conv5x5 benchmarks, we observe a significant increase in savings as the latency constraint increases. This trend represents the general case where the previous approach has difficulties finding an optimal schedule with larger operation counts. There are exceptions to this trend as seen in the linsor benchmark where the previous approach performs better than the FD-FTA heuristic under most latency constraints. In fact, the FD-FTA heuristic does progressively worse as the latency constraint increases.

The second trend of increased savings with increased operation count can be attributed to the previous approach's iterative approach and its randomness in the scheduling algorithm. Since that approach largely relies on finding an optimal schedule randomly over many iterations, that approach is much more likely to find an optimal schedule in small DFGs rather than in larger DFGs given good starting parameters. As a result, as the size of the DFG increases, the previous approach finds less optimal solutions and encounters its exit condition much earlier. Although a design could modify the starting parameters to find a better solution at different sizes, such an approach requires much fine-tuning and is not a scalable solution. In contrast, the FD-FTA heuristic uses a global scheduling approach that balances operation concurrency over all possible cycles which allows the heuristic to scale with the size of the DFG at the expense of additional complexity.

5.4. Heuristic Execution Time Comparison. This section compares the average heuristic execution times of the FD-FTA heuristic with the previous approach. In this context, execution time refers to the duration of the scheduling and binding process and does not include the placement and routing of the design on a FPGA. We provide this comparison to demonstrate the scalability of these heuristics, which may limit the practical use cases of a heuristic. For FPGAs, long execution times may be an acceptable tradeoff due to placement and routing generally dominating compiling times. Relatively short execution times may make a heuristic amenable to fast recompilation in areas such as FPGA overlays [33].

Figure 8 presents the results of this experiment comparing the execution time of the two approaches with a 100% EC constraint and a 2x the normalized latency constraint. The figure is arranged such that the horizontal axis represents benchmarks arranged by operation count from smallest to largest and that the vertical axis represents execution time in milliseconds on a logarithmic scale. It should be noted that each tick of the horizontal axis does not represent the same increase in operation count between benchmarks. Similarly, benchmarks of the same operation count may vary in execution for both heuristics based on the overall DFG structure.

Generally, the results of this experiment match our expectations on execution time. The figure shows that, for each benchmark, the previous approach's execution time is orders of magnitude longer than the FD-FTA heuristic with an average speedup of 96x, a median speedup of 40x, and a max speedup of 570x in the $fft16$ benchmark. Our switch to the proposed FD-FTA heuristic was intended to address the previous approach's long execution time.

Based on the experiments on resource savings, these results suggest that both heuristics and the TMR-RTL approach have Pareto-optimal solutions within the design space for execution times, resource requirements, and latency constraints. Although the TMR-RTL approach does not require any scheduling and binding, results from Table 3 indicate a much smaller design may be achievable with the other

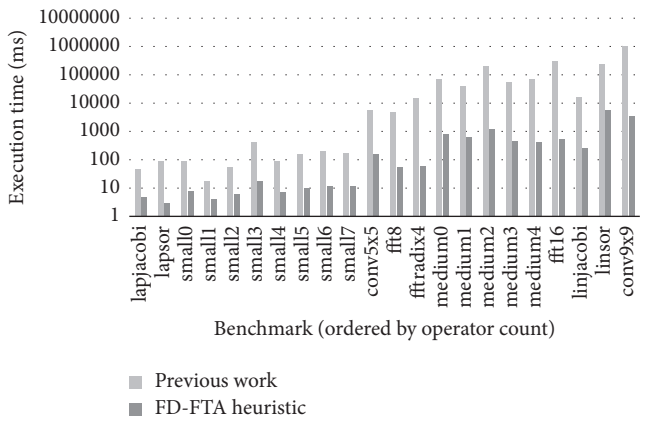

Figure 8: Execution times of the three approaches in the benchmark set under a 2x normalized latency constraint and a 100% EC constraint.

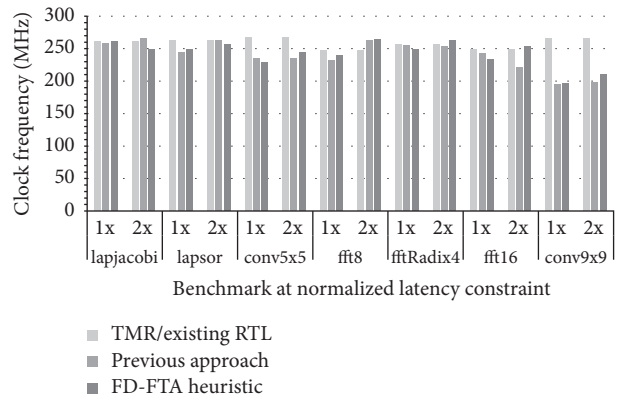

Figure 9: Clock frequency of the three approaches in a subset of nonsynthetic benchmarks under 1x and 2x normalized latency constraints and a 100% EC constraint.

approaches, especially at a larger latency constraint. Similarly, the approach of previous work may take vastly longer than other approaches but may achieve significantly smaller designs for latency constraints at the minimum-possible latency at a negligible difference in execution time for small DFGs. Notably, the FD-FTA heuristic achieves significantly more savings than the previous work at increased latency constraints and on larger DFGs.

5.5. Clock Results. This section compares the clock frequencies of designs generated by the proposed heuristic, the TMR-RTL approach, and the previous approach. Since increased resource sharing can increase the size of input multiplexers and potentially the critical path, we include this experiment to explore the impact of each approach on a design's clock frequency. Since synthesis tools perform placement and routing with a pseudorandom process, we determine an approach's clock frequency over multiple compilation runs through a script that does a binary search over the possible clock-constraint space and determines the theoretical max clock frequency based on a target frequency and slack values. Due to the lengthiness of this process, we provide this comparison only on a subset of nonsynthetic benchmarks and on a reduced set of constraints. For constraints, we test each selected benchmark at 1x and 2x normalized latency constraint with 100% EC constraint. Since the TMR-RTL approach does not rely on the latency constraint, we repeat the value in both latency constraints for comparison purposes.

To obtain these results, we convert the solution's netlist provided the C++ code into RTL code through scripts. We then use Vivado 2015.2 for compilation and for determining clock frequencies after placement and routing. For resources, we use IP cores from Xilinx for each respective resource and configure them to be implemented in LUTs and with single-cycle latencies.

Figure 9 shows the results of this experiment. In this figure, each approach has a bar for each benchmark and normalized latency constraint pairing. On the horizontal axis, the results are clustered by benchmark and then by normalized latency constraint relative to the minimum-possible latency. On the vertical axis, the clock frequency is reported in MHz.

For each benchmark and latency constraint pairing, Figure 9 presents similar results for each approach. Compared to the TMR-RTL approach, the previous approach and the FD-FTA heuristic had an average clock degradation of 7.08% and 6.03%, respectively. This clock overhead is unexpectedly small since sharing increases the steering logic and the FD-FTA heuristic and previous approach incorporate much more sharing than the TMR-RTL approach. Upon further analysis of the timing reports, each approach had different critical paths. For the TMR-RTL approach, the critical path was generally a path from an input register to the output register of a multiplier. For the other approaches, the critical path was usually a path from the output register of one resource to the output register of a multiplier. These results suggest that the larger routing problem associated with using more resources in the TMR-RTL approach had a similar impact on a design's clock frequency as the steering logic for the other approaches. The one exception to this trend was the *Conv9x9* benchmark where the TMR-RTL approach had much higher clock frequencies than the other two approaches. In terms of increasing the latency constraint, both the FD-FTA heuristic and the previous approach generally showed minor increases in clock frequency.

6. Conclusions

This paper introduced the Force-Directed Fault-Tolerance-Aware (FD-FTA) heuristic to solve the minimum-resource, latency- and error-constrained scheduling and binding problem. Compared to TMR applied to existing RTL (TMR-RTL) and to a previous approach, this heuristic provides attractive tradeoffs by efficiently performing redundant operations on shared resources. The FD-FTA heuristic improves upon previous work with improved scheduling and binding algorithms for better scalability on larger benchmarks and with increased latency constraints. Compared to TMR-RTL implementations, the FD-FTA heuristic had average savings of 34%

and displayed average savings of 47% at a 2x normalized latency constraint. Although skewed by small benchmarks which have little opportunities for sharing, the FD-FTA heuristic had savings up to 80% under a 2x normalized latency constraint, with most savings occurring in the larger benchmarks. Compared to the previous approach, the FD-FTA heuristic had average savings of 19% and displayed average savings of 22% at a 2x normalized latency constraint. The FD-FTA's comparisons with the previous approach are also skewed by small benchmarks and showed up to 74% resource savings under a 2x normalized latency constraint with most savings occurring in the larger benchmarks. Additionally, the FD-FTA showed a 96x average heuristic execution time improvement compared to the previous approach and produced FPGA designs with a 6% average clock degradation compared to the TMR-RTL implementations. Future work includes support for pipelined circuits and for alternative strategies for FPGA fault-tolerance.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work is supported in part by the I/UCRC Program of the National Science Foundation, under Grant nos. EEC-0642422, IIP-1161022, and CNS-1149285.

References

[1] N. Wulf, A. D. George, and A. Gordon-Ross, "Memory-aware optimization of FPGA-based space systems," in *Proceedings of the IEEE Aerospace Conference (AERO '15)*, pp. 1–13, IEEE, March 2015.

[2] M. Wirthlin, "High-reliability FPGA-based systems: space, high-energy physics, and beyond," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 379–389, 2015.

[3] M. J. Wirthlin, "FPGAs operating in a radiation environment: Lessons learned from FPGAs in space," *Journal of Instrumentation*, vol. 8, no. 2, Article ID C02020, 2013.

[4] S. Golshan, H. Kooti, and E. Bozorgzadeh, "SEU-aware high-level data path synthesis and layout generation on SRAM-based FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 6, pp. 829–840, 2011.

[5] A. Shastri, G. Stitt, and E. Riccio, "A scheduling and binding heuristic for high-level synthesis of fault-tolerant FPGA applications," in *Proceedings of the 26th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '15)*, pp. 202–209, July 2015.

[6] A. Antola, V. Piuri, and M. Sami, "High-level synthesis of data paths with concurrent error detection," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 292–300, Austin, Tex, USA, November 1998.

[7] P. S. Ostler, M. P. Caffrey, D. S. Gibelyou et al., "SRAM FPGA reliability analysis for harsh radiation environments," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3519–3526, 2009.

[8] O. Héron, T. Arnaout, and H.-J. Wunderlich, "On the reliability evaluation of SRAM-based FPGA designs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 403–408, August 2005.

[9] O. Boncalo, A. Amaricai, C. Spagnol, and E. Popovici, "Cost effective FPGA probabilistic fault emulation," in *Proceedings of the 32nd NORCHIP Conference (NORCHIP '14)*, pp. 1–4, October 2014.

[10] A. Janning, J. Heyszl, F. Stumpf, and G. Sigl, "A cost-effective FPGA-based fault simulation environment," in *Proceedings of the 8th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC '11)*, pp. 21–31, September 2011.

[11] J. McCollum, "ASIC versus antifuse FPGA reliability," in *Proceedings of the IEEE Aerospace Conference*, pp. 1–11, March 2009.

[12] K. S. Morgan, D. L. McMurtrey, B. H. Pratt, and M. J. Wirthlin, "A comparison of TMR with alternative fault-tolerant design techniques for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2065–2072, 2007.

[13] C. Bolchini, A. Miele, and M. D. Santambrogio, "TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs," in *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems, DFT 2007*, pp. 87–95, September 2007.

[14] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for FPGA designs using triple modular redundancy," in *Proceedings of the 18th Annual ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '10)*, pp. 249–258, ACM, Monterey, Calif, USA, February 2010.

[15] P. G. Paulin and J. P. Knight, "Scheduling and binding algorithms for high-level synthesis," in *Proceedings of the 26th ACM/IEEE conference*, pp. 1–6, June 1989.

[16] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[17] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[18] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. 2, pp. 1258–1263, March 2005.

[19] X. Chen, W. Yang, M. Zhao, and J. Wang, "HLS-based sensitivity-inductive soft error mitigation for satellite communication systems," in *Proceedings of the 22nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS '16)*, pp. 143–148, July 2016.

[20] M. B. Hammouda, P. Coussy, and L. Lagadec, "A unified design flow to automatically generate on-chip monitors during high-level synthesis of hardware accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 384–397, 2017.

[21] S. Hadjis, A. Canis, J. H. Anderson et al., "Impact of FPGA architecture on resource sharing in high-level synthesis," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '12)*, pp. 111–114, ACM, Monterey, Calif, USA, February 2012.

[22] A. F. Dos Santos, L. A. Tambara, F. Benevenuti, J. Tonfat, and F. L. Kastensmidt, "Applying TMR in hardware accelerators generated by high-level synthesis design flow for mitigating multiple bit upsets in SRAM-based FPGAs," in *Applied Reconfigurable Computing*, pp. 202–213, Springer, 2017.

[23] H. Quinn, D. Roussel-Dupre, M. Caffrey et al., "The cibola flight experiment," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, article no. 3, 2015.

[24] A. Orailoglu and R. Karri, "Automatic synthesis of self-recovering VLSI systems," *IEEE Transactions on Computers*, vol. 45, no. 2, pp. 131–142, 1996.

[25] K. Kyriakoulakos and D. Pnevmatikatos, "A novel SRAM-based FPGA architecture for efficient TMR fault tolerance support," in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 193–198, August 2009.

[26] T. Inoue, H. Henmi, Y. Yoshikawa, and H. Ichihara, "High-level synthesis for multi-cycle transient fault tolerant datapaths," in *Proceedings of the IEEE 17th International On-Line Testing Symposium (IOLTS '11)*, pp. 13–18, July 2011.

[27] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[28] F. J. Kurdahi and A. C. Parker, "Real: a program for register allocation," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 210–215, June 1987.

[29] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proceedings of the 8th Design Automation Workshop (DAC '71)*, pp. 155–169, ACM, Atlantic City, NJ, USA, June 1971.

[30] A. Canis, J. Choi, M. Aldham et al., "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*, pp. 33–36, ACM, Monterey, Calif, USA, March 2011.

[31] J. A. Vite-Frias, R. D. J. Romero-Troncoso, and A. Ordaz-Moreno, "VHDL core for 1024-point radix-4 FFT computation," in *Proceedings of the proceedings of the IEEE International Conference on Reconfigurable Computing and FPGAs (RECONFIG '05)*, pp. 4–24, Puebla, Mexico, September 2005.

[32] J. Hu, *Solution of partial differential equations using reconfigurable computing [PhD. thesis]*, The University of Birmingham, 2010.

[33] J. Coole and G. Stitt, "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts," *IEEE Micro*, vol. 34, no. 1, pp. 42–53, 2014.