A Scheduling and Binding Heuristic for High-Level Synthesis of Fault-Tolerant FPGA Applications

Aniruddha Shastri, Greg Stitt, and Eduardo Riccio
Department of Electrical and Computer Engineering
University of Florida
Gainesville, FL, USA
aniruddha.shastri@gmail.com, gstitt@ece.ufl.edu, eduardo.riccio@gmail.com

***Abstract*—Space computing systems commonly use field-programmable gate arrays to provide fault tolerance by applying triple modular redundancy (TMR) to existing register-transfer-level (RTL) code. Although effective, this approach has a 3x area overhead that can be prohibitive for many designs that often allocate resources before considering effects of redundancy. Although a designer could modify existing RTL code to reduce resource usage, such a process is time consuming and error prone. Integrating redundancy into high-level synthesis is a more attractive approach that enables synthesis to rapidly explore different tradeoffs at no cost to the designer. In this paper, we introduce a scheduling and binding heuristic for high-level synthesis that explores tradeoffs between resource usage, latency, and the amount of redundancy. In many cases, an application will not require 100% error correction, which enables significant flexibility for scheduling and binding to reduce resources. Even for applications that require 100% error correction, our heuristic is able to explore schedules that sacrifice latency for reduced resources, and typically save up to 47% when relaxing the latency up to 2x. When the error constraint is reduced to 70%, our heuristic achieves typical resource savings ranging from 18% to 49% when relaxing the latency up to 2x, with a maximum of 77%. Even when comparing with optimized RTL designs, our heuristic uses up to 61% fewer resources than TMR.**

I. Introduction

Space computing systems increasingly rely on field-programmable gate arrays (FPGAs) to meet performance and power constraints that cannot be met by other computing technologies [1]. One unique challenge in such systems is the requirement for fault tolerance due to radiation-induced single-event upsets (SEUs). In addition to providing performance and power advantages, FPGAs also provide a convenient way of protecting against such upsets by using redundant logic.

One common form of redundancy is triple modular redundancy (TMR), which replicates a register-transfer-level (RTL) circuit three times and then votes on the outputs to identify and correct errors. When combined with scrubbing (i.e., reconfiguring a faulty resource), TMR provides an effective level of fault tolerance for many applications.

One key disadvantage of TMR is a $3\times$ increase in resources, which can be prohibitive for some applications, or can significantly increase cost by requiring a larger FPGA. One option for reducing TMR overhead is to manually explore different schedules for an RTL circuit to use fewer resources at the expense of performance. However, such exploration in RTL code is time consuming and error prone. For example, in many situations, designers apply TMR to existing IP [2] that is difficult to modify.

Alternatively, if designers specify their application using high-level code, high-level synthesis can perform such exploration without any effort required by the application designer. Previous work has introduced high-level synthesis techniques for generating fault-tolerant circuits [3], [4], but those studies focused largely on ASICs, or where SEUs tend to cause transient errors [4]–[6]. Because FPGAs are largely SRAM devices, errors caused by SEUs are often semi-permanent because the errors will exist indefinitely until after scrubbing.

In this paper, we introduce an FPGA-specialized heuristic for scheduling and binding that enables high-level synthesis to identify attractive tradeoffs between latency and area while also applying redundancy to correct errors. This heuristic is based on the observation that although detecting errors is critical, correcting all errors is not always as necessary. Therefore, the heuristic includes in the exploration an *error-correction percentage* that can be specified as a design constraint. Although numerous studies have introduced heuristics for trading off latency and resources during high-level synthesis, our heuristic simultaneously considers latency, resources, and error-correction percentage. We show that our heuristic is able to save up to 77% of resources at $2\times$ latency, while still detecting all errors and correcting over 70% of errors. When the heuristic is constrained to below 100% error correction we observe up to an additional 24% savings when relaxing the latency by up to $2\times$. Furthermore, the reported results are pessimistic due to a decreased probability of SEUs for smaller circuits.

II. Related Work

Although there has been significant work on scheduling and binding problems for high-level synthesis [7]–[9], those studies do not consider fault tolerance or error correction. More recent work has integrated fault tolerance into high-level synthesis, but focuses on different problems than this paper. For example, Safari [10] applies fault-tolerance measures after scheduling and binding, whereas our approach integrates fault-tolerance considerations into scheduling and binding. Bolchini [11] explores the design space of scheduling and binding with the goal of optimizing partial dynamic reconfiguration for designs that apply TMR, whereas we vary the quality of error correction to reduce resources. Sengupta [12] leverages bacterial foraging to perform binding based on power-performance tradeoffs and targets transient faults, whereas our work focuses on variation

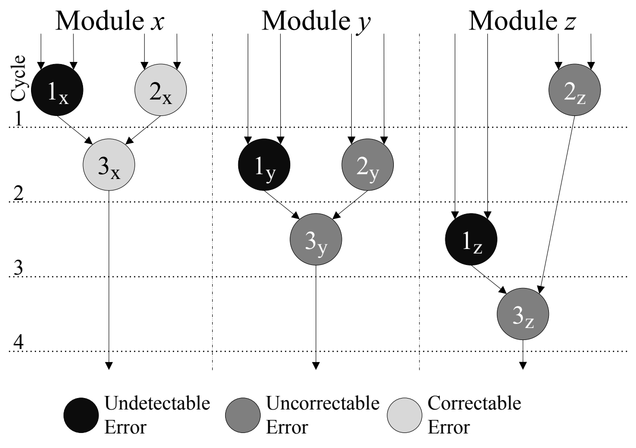


Fig. 1. An example binding onto three color-coded resources. For this binding, a single fault can result in an undetectable, uncorrectable (but detectable), or correctable error, depending on the resource. Each operation is identified as ID_{Module}.

in error-correction ability while maintaining error-detection capabilities for semi-permanent faults. Inoue [6] varies the quality of fault tolerance in ASICs, but considers multi-cycle transient SEU faults. Although complementary, our work focuses on semi-permanent faults due to the focus on FPGAs. Furthermore, some approaches consider only scheduling or binding, but not both (e.g., [12], [13]).

Previous work has also explored tradeoffs related to fault tolerance. Morgan et al. [14] compare TMR with other forms of redundancy and find TMR to be superior both in terms of reliability and cost, but that study did not evaluate Pareto-optimal tradeoffs for lower error correction. Studies such as [3] and [15] consider tradeoffs between cost, latency, and reliability, but consider fault tolerance tradeoffs using resources that have different levels of built-in redundancy. The heuristic presented in this paper explores reduced reliability by sharing resources across redundant operations. Dynamic reconfiguration techniques using evolutionary algorithms provide some variation in reliability and cost, but often require high area and computational overhead [16]. The heuristic presented here seeks to address this overhead by minimizing area.

Other studies have investigated FPGA resources with built-in redundancy to combat SEUs [5], [17]. Our work complements such architectural strategies by enabling reduced fault-tolerance on resources that lack built-in redundancy.

Many studies have investigated using partial reconfiguration to reconfigure the part of the FPGA affected by a SEU in [13], [18]–[20], which can be combined with our work to improve the fault tolerance of the overall design.

Other work has focused on how to prioritize the application of redundancy to resources that are more important and/or are more susceptible to faults [21], [22]. Siozios et al. [23] discuss Adaptive TMR, which uses game theory to prioritize application of redundancy measures to critical sections of the design. Our work can be deployed to provide low fault tolerance to the less-critical sections identified by these methods, with low overhead. Previous work has also investigated optimizations due to voter placement [2]. Our work is complementary and could potentially integrate those voter placement strategies.

III. 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:

- *Fault*: A resource with an SEU-induced error.
- *Module*: One instance of the dataflow graph (DFG), analogous to a module in TMR.
- *Error*: Any fault where one or more of the three modules outputs an incorrect value.
- *Undetectable Error*: Any fault where all three modules output the same incorrect value.
- *Detectable Error*: Any fault where one or more modules output different values.
- *Uncorrectable Error*: Any fault where two or more modules output incorrect values.
- *Correctable Error*: Any fault where two or more modules output a correct value.
- *Error Correction % (EC%)*: The percentage of total possible errors that are correctable by a given solution.

Fig. 1 illustrates several example error types, where all operations with the same color are bound to a single resource. A fault in the black resource 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 (y and z) will produce incorrect values. A fault in the light-gray resource results in a correctable error because modules y and z 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 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:

$$\begin{array}{ll} \textit{Minimize} & \textsc{NumResources}(X) \\ \textit{Subject to} & \textsc{Latency}(X.S) \leq L \text{ and} \\ & \textsc{ErrorCorrection}\%(X.B) \geq E \text{ and} \\ & \textsc{ErrorDetection}\%(X.B) = 100\% \end{array}$$

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, resource-constrained scheduling followed by binding, which are both NP-hard problems [7], [9]. The

```
1:  function FTA($D, L, E$)
2:      $i \leftarrow 0$
3:      $X_c \leftarrow X_p \leftarrow X_b \leftarrow \emptyset$
4:      $D_{FT} \leftarrow$ TRIPLICATE(D)
5:      while STOP($X_p, X_b, i$) = false do
6:          if IsPow2($i$) then
7:              $X_p \leftarrow X_b$                ▷ Save previous best solution
8:          $S \leftarrow$ SCHEDULE($D_{FT}, L$)
9:          $B \leftarrow$ BIND($S, E$)
10:         $X_c \leftarrow \{S, B\}$                ▷ Store current solution
11:         if Q($X_c$) $>$ Q($X_b$) then
12:             $X_b \leftarrow X_c$                 ▷ Update best solution
13:         $i \leftarrow i + 1$
14:     return $X_b$                         ▷ Return best solution found

15: function STOP($X_p, X_b, i$) ▷ Check for stopping condition
16:     if $i \leq minTests$ then
17:         return false
18:     else if IsPow2($i$) and Q($X_b$) $\leq$ Q($X_p$)$\times Z$ then
19:         return true                      ▷ Stopping condition reached
20:     else
21:         return false
```

Fig. 2. Fault-Tolerance-Aware (FTA) Heuristic

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 practical definition of this problem would minimize FPGA resources (e.g., lookup tables, DSP units), as opposed to the number of coarse-grained resources (e.g., adders, multipliers), due to different operations requiring significantly different numbers and types of FPGA resources. The problem could easily be extended by simply replacing each coarse-grained resource by the equivalent FPGA resources. However, for ease of explanation, we present the problem and heuristic in terms of numbers of coarse-grained resources. Similarly, we could expand this definition to handle constraints on initiation interval, performance, etc., but we focus on latency and number of resources to make our experiments more comparable to previous work [6], [19]. We plan to investigate pipelined implementations in future work.

We assume that scrubbing occurs frequently enough so there cannot be more than one faulty resource at a time, which is often true for common SEU rates [1]. 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 [4]–[6]. We assume the presence of an implicit voter at the output of the modules, potentially using strategies from [2].

One potential challenge with error correction is the possibility of two modules producing incorrect outputs 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 use cases [6]. 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.

IV. Fault-Tolerance-Aware (FTA) Heuristic

To solve the problem defined in the previous section, we introduce the *Fault-Tolerance-Aware (FTA) Heuristic* as shown in Fig. 2. The heuristic takes as input a DFG D, a latency constraint L, and an error constraint E and creates a schedule and binding of D that attempts to minimize resources while meeting constraints. Note that before running this heuristic, high-level synthesis would decompose the entire application into multiple dataflow graphs, consisting of dataflow graphs from different control states and decomposed dataflow graphs for efficient voter placement [11], [24]. High-level synthesis would then run this heuristic on each dataflow graph separately.

The heuristic initially triplicates D and then iteratively explores potential solutions (lines 5-13) by performing scheduling (Section IV-A) and binding (Section IV-B). During each iteration, the heuristic compares the quality (i.e., number of resources returned from function Q(X)) of the current solution X_c with the best solution so far X_b, and updates X_b when necessary (lines 11-12). Note that solutions not meeting the constraints will not be output by the scheduler or binder.

One scheduling challenge that is unique to handling error correction is that the scheduler is unaware of the EC% until after binding. Although a scheduler could potentially use a cost function to estimate the results of binding, such a function is not obvious. Instead, the heuristic introduces randomness into each schedule using *Random Non-Zero Slack List Scheduling*, which we introduce in Section IV-A, in a way that meets latency constraints. For each schedule, the heuristic then performs *Singleton-Share Binding*, discussed in Section IV-B, to map operations onto resources in a way that meets error constraints while also reducing resources.

The heuristic stops exploring new schedules and bindings when certain stopping conditions, determined by the STOP() function, become true. The heuristic explores solutions in multiple phases, where each phase explores $2\times$ more solutions than the previous phase. Anytime a new phase begins, the current best solution is saved as the solution from the previous phase X_p (lines 6-7). As shown in lines 18-19, the stopping condition occurs at the end of a phase (i.e., when i is a power of 2) when the quality of the best solution is less than a user-definable percentage Z better than the solution from the previous phase. In other words, the heuristic completes a phase and then checks if there was a significant improvement compared to the previous phase. If so, the heuristic searches $2\times$ more solutions until no significant improvement is found. To ensure that noise from small numbers of tests does not cause the heuristic to end early, the stopping condition includes a user-configurable minimum numbers of tests (shown as $minTests$ in line 16).

A. Random Non-Zero Slack List Scheduling

Traditional scheduling methods such as list scheduling [7] deterministically create a single schedule that specifies the

```
function SCHEDULE($D, L$)
    $S_l \leftarrow \emptyset$                                ▷ Scheduled DFG
    $S_a \leftarrow$ ALAPSCHEDULE($D, L$)
    $L_c \leftarrow \emptyset$                    ▷ List of candidate operations
    for $r \in$ RESOURCETYPES(D) do
        $Usage_r \leftarrow 1$
    while $D.Ops - S_l.Ops \neq \emptyset$ do
        for $r \in$ RESOURCETYPES(D) do
            for $c \leftarrow 1, \ldots, L$ do ▷ Meet latency constraint
                $count \leftarrow 0$
                $L_c \leftarrow$ GETCANDIDATELIST(D,r)
                for $op \in L_c$ s.t. SLACK($op, S_a, c$) $= 0$ do
                    $S_l.Ops \leftarrow S_l.Ops \cup \{op\}$
                    $S_l.Cycle[op] \leftarrow c$
                    $count \leftarrow count + 1$
                    if $count > Usage_r$ then
                        $Usage_r \leftarrow Usage_r + 1$
                for $op \in L_c$ s.t. SLACK($op, S_a, c$) $\neq 0$ do
                    if RANDOM() $> 0.5$ then
                        if $count < Usage_r$ then
                            $S_l.Ops \leftarrow S_l.Ops \cup \{op\}$
                            $S_l.Cycle[op] \leftarrow c$
                            $count \leftarrow count + 1$
    return $S_l$                              ▷ Return scheduled DFG
```

Fig. 3. Random Non-Zero Slack List Scheduling Heuristic

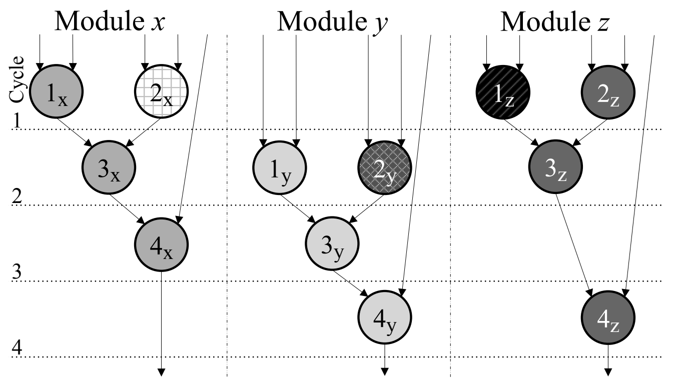
number of required resources independently from binding. When considering error correction, such an approach does not work because the EC% is not known until after binding.

To adapt existing schedulers to better support error correction, our heuristic uses an extended version of minimum-resource, latency-constrained list scheduling that we refer to as *Random Non-Zero Slack List Scheduling*, which is shown in Fig. 3. For brevity, we omit a complete explanation of traditional list scheduling and refer the reader to previous work [7]. Like traditional list scheduling, our scheduler initially uses an ALAP schedule to determine the *slack* of each operation, which is the remaining number of cycles before the operation has to be scheduled without violating latency constraints. Similarly, our heuristic always schedules operations that have zero slack (lines 12-17).

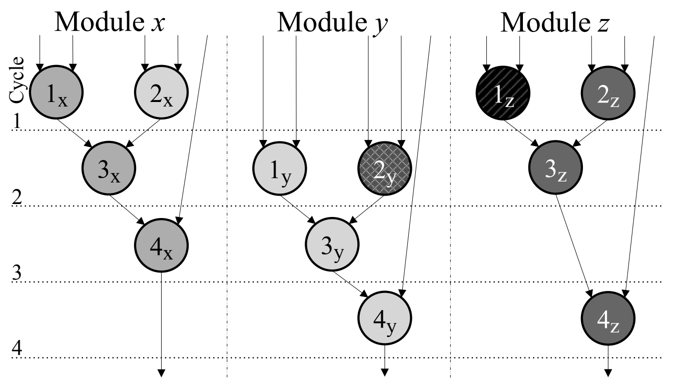
Our scheduler differs from traditional list scheduling in the way that non-zero slack candidates are handled. Traditional list scheduling only schedules non-zero slack candidates when the number of resources does not increase. As shown in lines (18-23), our scheduler injects randomness into the schedule by scheduling non-zero slack candidates with a probability of 50%. When applied to the triplicated dataflow graph, this randomness allows for each module to have a different schedule, further enabling binding to share resources across modules, which reduces resources compared to TMR.

B. Singleton-Share Binding

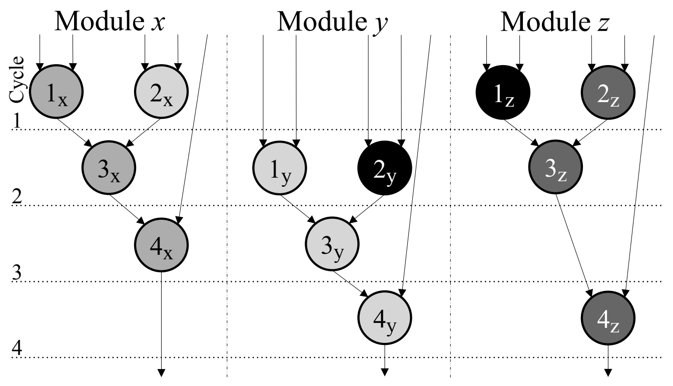
Given a schedule and an error constraint, binding is responsible for mapping operations from the schedule onto a resource, in a way that minimizes resources while meeting the error constraint. Traditional binding algorithms typically directly use the resource requirements specified by the scheduler,



(a) Binding after stage 1, with patterned singleton bindings.



(b) Stage 2 intermediate binding at line 35 of binding algorithm.



(c) Binding after stage 2.

Fig. 4. Sample binding for a latency constraint of 4 cycles and error constraint of 50%. In the sub-figures, resources are differentiated by the color of the node. The patterned nodes in (a) and (b) represent singleton bindings. According to Equation 4, at most 2 resources can be shared across two modules. In (b), 2_x, 1_y, 3_y and 4_y are selected to share a resource, resulting in an EC% of 80%. Merging the singleton bindings of 2_y and 1_z results in binding (c), with an EC% of 50%.

without considering reduction in fault tolerance by additional sharing of resources across modules. To deal with this issue, we introduce the *Singleton-Share Binding Heuristic*.

For ease of presentation, the following explanation uses the simplified problem definition of minimizing the total number of coarse-grained resources. To handle FPGA resources instead, the heuristic would simply translate each coarse-grained resource into FPGA resources, and then prioritize sharing of coarse-grained resources that use the most FPGA resources.

Singleton-Share Binding performs binding in two stages. The first stage binds resources within each separate module using a common algorithm based on clique partitioning [8]. At the end of stage 1, the heuristic has bound each operation onto a minimum number of resources within each module, while maintaining 100% error detection and error correction because no operations are shared across modules. Fig. 4(a) demonstrates an example stage 1 binding, where the six different colors represent six different resources.

The second stage tries to reduce resources by sharing resources across operations from different modules. The heuristic selects a *singleton*, which is a resource that executes a single operation, and then shares that operation with a resource used in a different module. After performing this share, the singleton resource is no longer needed and can be removed. Fig. 4(b) demonstrates a singleton share of operation 2_x onto the resource used by operations 1_y, 3_y, and 4_y, in which case the original resource used by 2_x is eliminated. However, this share also reduces the EC% from 100% to 80% because this binding now has one uncorrectable error out of five total possible errors from the five remaining resources.

The main challenge of the heuristic is that the sharing cannot allow the EC% to fall below the error constraint. Because the number of uncorrectable errors equals the resources that have been shared across two modules, the condition for the EC% constraint to be satisfied is as shown in (1),

$$\frac{resources_{used} - resources_{mShared}}{resources_{used}} \times 100\% \geq E \qquad (1)$$

which can can be reduced to (2):

$$resources_{mShared} \leq resources_{used} \times (1 - {}^{E}\!/_{100}) \qquad (2)$$

Each time the heuristic eliminates a singleton resource by sharing the corresponding operation, the number of resources shared across modules increases by one. Hence, for i such shares performed on a stage 1 initial binding that uses $usage_{stg1}$ resources, (2) is equivalent to (3).

$$i \leq (usage_{stg1} - i) \times (1 - {}^{E}\!/_{100}) \qquad (3)$$

Since the $limit$ on the number of singleton shares is an integer, i from (3) is equivalent to $limit$ in (4). Using (4), the heuristic can calculate at the outset how many singleton shares can be performed before the error constraint is no longer satisfied.

$$limit \leq \lfloor usage_{stg1} \times {}^{100-E}\!/_{200-E} \rfloor \qquad (4)$$

The full Singleton-Share Binding Heuristic is shown in Fig. 5. The heuristic initially performs the stage 1 binding (line 2) and then passes that binding, along with the schedule and error constraint to the stage 2 binder. Stage 2 initially determines the number of resources used by the stage 1 binding and calculates the maximum amount of sharing without violating the error constraint (lines 11-12). In the figure, every b corresponds to a set of operations mapped to a single resource (i.e., the binding for that resource). B represents a binding for multiple resources as a set of these b sets. Line 13 creates a set of all operations that were mapped onto singleton resources in stage 1. Lines 14-15 initialize the stage 2 binding. The loop

```
1:  function BIND($S, E$)
2:      $B_{stg1} \leftarrow$ STAGE1($S$)
3:      $B \leftarrow$ STAGE2($S, E, B_{stg1}$)
4:      return $B$

5:  function STAGE1($S$)                ▷ Bind modules independently
6:      $B_{stg1} \leftarrow \emptyset$
7:      for $S_i \in$ INDIVIDUALMODULES($S$) do
8:          $B_{stg1} \leftarrow B_{stg1} \cup$ CLIQUEPARTITIONING($S_i$)
9:      return $B_{stg1}$

10: function STAGE2($S, E, B_{stg1}$)       ▷ Bind across modules
11:     $usage_{stg1} \leftarrow$ TOTALRESOURCEUSAGE($B_{stg1}$)
12:     $limit \leftarrow \lfloor usage_{stg1} \times {}^{100-E}/_{200-E} \rfloor$         ▷ From (4)
13:     $B_s \leftarrow \{b_s \in B_{stg1} \mid |b_s| = 1\}$
14:     $B_{stg2} \leftarrow B_{stg1}$                ▷ Initialize binding solution
15:     $mSharing \leftarrow 0$       ▷ Resouces shared across modules
16:     for $\{b_s\} \in B_s$ do     ▷ Loop through list of singletons
17:         if $mSharing = limit$ then
18:             return $B_{stg2}$              ▷ EC constraint reached
19:         for $b \in B_{stg2} - B_s$ do   ▷ Loop through resources
20:             if RESTYPE($b$)=RESTYPE($b_s$) then
21:                 $m_b \leftarrow$MODULES($b$)
22:                 $m_{b_s} \leftarrow$MODULE($b_s$)
23:                 if $|m_b| = 1$ or $m_{b_s} \in m_b$ then
24:                     $isResourceFound \leftarrow True$
25:                     for $c \in$ CYCLES($b, S$) do
26:                         if $c = S.Cycle[b_s]$ then
27:                             $isResourceFound \leftarrow False$
28:                     if $isResourceFound = True$ then
29:                         if $m_{b_s} \notin m_b$ then
30:                             $mSharing \leftarrow mSharing + 1$
31:                         $b \leftarrow b \cup \{b_s\}$        ▷ Share singleton
32:                         $B_{stg2} \leftarrow B_{stg2} - \{b_s\}$
33:                         $B_s \leftarrow B_s - \{b_s\}$
34:                         break   ▷ Try next singleton in $B_s$
35:     for $\{b_i\}, \{b_j\} \in B_s$ s.t. $S.Cycle[b_i] \neq S.Cycle[b_j]$ do
36:         if RESTYPE($b_i$)=RESTYPE($b_j$) then
37:             if $mSharing = limit$ then
38:                 return $B_{stg2}$          ▷ EC constraint reached
39:             if MODULE($b_i$)$\neq$MODULE($b_j$) then
40:                 $mSharing \leftarrow mSharing + 1$
41:             $B_{stg2} \leftarrow B_{stg2} \cup \{b_i, b_j\}$    ▷ Merge singletons
42:             $B_{stg2} \leftarrow B_{stg2} - \{b_i\} - \{b_j\}$
43:             $B_s \leftarrow B_s - \{b_i\} - \{b_j\}$
44:     return $B_{stg2}$                  ▷ Return binding solution
```

Fig. 5. Singleton-Share Binding Heuristic

from line 16 to 34 initially selects an operation b_s mapped onto a singleton resource (line 16). Next, the heuristic checks to see if more sharing can be done. If so, the loop from line 19 to 34 considers binding b_s onto a non-singleton resource used by a different set of operations b. If all of these operations are the same type (line 20), and if the b operations are either all in one module or share the same module as b_s (line 23), then the heuristic considers binding b_s to the resource used by b. The condition in line 23 ensures that error detection capability is maintained. The loop at line 25 checks to ensure that none

TABLE I. BENCHMARK SUMMARY

Benchmark	*Description*
conv5x5	5x5 convolution kernel.
fft8	8-point radix-2 DIT FFT based on butter-fly architecture.
fft16	16-point radix-2 DIT FFT based on butter-fly architecture.
fftrad4	4-point radix-4 FFT based on dragon-fly architecture. Efficiently decomposes complex operations into real operations [25].
linsor	Single iteration kernel to solve a system of linear equations ($Ax = B$) in 5 variables, using successive-over-relaxation (SOR) method [26].
linjacobi	Single iteration kernel to solve a system of linear equations in 5 variables, using Jacobi method [26].
lapsor	Single iteration kernel to solve Laplace's equation ($\nabla^2 f = 0$), using successive-over-relaxation (SOR) method [26].
lapjacobi	Single iteration kernel to solve Laplace's equation using Jacobi method [26].
dfg0-dfg7	Eight randomly generated DFGs.

of the operations in b occur at the same time as b_s, otherwise sharing is not possible. If sharing is possible, the heuristic moves operation b_s onto the resource used by b (lines 28-34) and updates the amount of sharing across modules if necessary (lines 29-30).

The heuristic repeats this process until there are no more non-singleton resources that can share operations from remaining singletons. Fig. 4(b) provides an example of this situation, where the heuristic cannot share 2_y and 1_z from their singleton resources with any other resource. At this point, the heuristic considers combining operations from multiple singletons (lines 35-43) as long as the operations are scheduled in different cycles. Fig. 4(c) demonstrates this process by mapping operation 2_y onto the singleton resource used by 1_z.

V. EXPERIMENTS

In this section, we evaluate the proposed heuristic by recording the minimum resource usage observed for a variety of input DFGs given different latency and error constraints. Section V-A discusses the experimental setup. Section V-B evaluates resource savings compared to existing RTL code not optimized for TMR. Section V-C evaluates similar experiments but compares to RTL code optimized for TMR.

A. Experimental Setup

To evaluate the heuristic we implemented an analysis tool in C#, combined with a PowerShell script, to automate the experiments on a 64-bit Windows 7 Enterprise machine.

We evaluate 16 DFG benchmarks, which are summarized in Table I. To represent signal-processing applications, we modeled 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 [25]. For the radix-2 FFTs, we use resources capable of directly performing complex operations. To represent fluid-dynamics and similar applications, we modeled two DFGs that solve 5-dimensional linear equations ($Ax = B$) using the Jacobi iterative method and the successive over-relaxation (SOR) iterative method [26]. We also modeled two DFGs that solve Laplace's equation ($\nabla^2 f = 0$) using the Jacobi and SOR methods. We also complemented these real benchmarks with eight synthetic benchmarks (dfg0-dfg7) that we created using randomly generated DFGs.

We use two baselines for different usage scenarios:

1) TMR Applied to Existing RTL (Section V-B): Designers commonly achieve fault tolerance by applying TMR to existing RTL code. As a result, that RTL code might use a number of resources that is acceptable without redundancy, but results in significant overhead after applying TMR. To approximate this usage, we assume that the RTL uses an ASAP schedule, which we have observed to be common in RTL implementations.

2) TMR Applied to Optimized RTL (Section V-C): For usage scenarios where a designer is willing to optimize an RTL implementation to reduce resources before applying TMR, we approximate that optimized RTL by using a schedule from minimum-resource, latency-constrained list scheduling [7].

To simplify discussion of results, we present overall resource savings instead of savings of each type of resource, which varies for each example. Resource savings of 0% corresponds to no improvement, whereas resource savings of 75% corresponds to resource usage being a quarter of the baseline's total number of resources. Note that by this definition, the theoretical maximum resource savings achievable is always less than 100%. We consider four types of operations - addition, subtraction, multiplication and division - mapped to three types of resources - adder, multiplier, and divider. Each addition or subtraction operation is mapped to an adder resource, each multiplication operation is mapped to a multiplier resource, and each division operation is mapped to a divider resource. The heuristic could be extended to optimize for a particular type of FPGA resource (e.g. LUTs) by considering FPGA requirements of each type of resource, and could be integrated with glue logic estimation approaches [27].

B. Comparison with TMR applied to existing RTL

Table II presents resources savings of the FTA heuristic compared to existing RTL code using TMR. The rows of the table correspond to different benchmarks, and the columns correspond to different latency constraints up to $2\times$ the minimum possible latency. The latency constraint is normalized by each DFG's minimum possible latency. Each table entry shows two interpolated savings results for an error correction constraint of 100% and 70+% respectively, where the latter refers to the average of the savings observed for error-correction values varying between 70% and 99%. Interpolation is required because for DFGs of different depths, variation in latency in steps of 1 cycle does not correspond to variation in the normalized latency constraint in steps of exactly $0.1\times$.

Even for the minimum-latency schedule, the FTA heuristic was able to save between 16% and 18% on average for the two different error constraints. As the latency constraint increased to $2\times$ the minimum latency, the heuristic saved up to an average of 47% and 49% for each error correction constraint. The difference in savings with the two error correction constraints reached a maximum value of 24% for the dfg1 benchmark at $1.5\times$ latency relaxation.

TABLE II. % RESOURCE SAVINGS OF FTA HEURISTIC COMPARED TO EXISTING RTL USING TMR FOR 100% AND 70+% EC

Benchmark	**1.0×**		**1.1×**		**1.2×**		**1.3×**		**1.4×**		**1.5×**		**1.6×**		**1.7×**		**1.8×**		**1.9×**		**2.0×**	
	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%	100%	70+%
conv5x5	19%	19%	33%	34%	45%	46%	51%	57%	53%	53%	52%	60%	55%	61%	58%	62%	59%	63%	60%	64%	59%	64%
fft8	6%	9%	9%	13%	11%	17%	14%	22%	17%	27%	24%	33%	31%	38%	37%	40%	42%	42%	41%	45%	39%	48%
fft16	8%	10%	9%	11%	11%	14%	13%	19%	20%	23%	25%	28%	28%	32%	29%	36%	37%	43%	44%	48%	47%	52%
fftrad4	0%	0%	7%	9%	13%	18%	16%	21%	18%	24%	21%	26%	23%	28%	23%	31%	23%	33%	30%	35%	36%	37%
linsor	67%	69%	69%	72%	71%	74%	73%	75%	72%	75%	74%	76%	73%	76%	75%	77%	74%	77%	75%	77%	75%	77%
linjacobi	28%	28%	34%	37%	39%	43%	42%	47%	46%	50%	50%	51%	52%	52%	50%	55%	54%	56%	55%	57%	54%	57%
lapsor	0%	0%	0%	11%	0%	22%	17%	25%	33%	29%	33%	30%	33%	31%	33%	32%	33%	33%	33%	32%	33%	31%
lapjacobi	0%	0%	9%	10%	18%	19%	22%	25%	22%	28%	22%	30%	26%	34%	31%	37%	33%	37%	33%	35%	33%	33%
dfg0	10%	13%	16%	18%	22%	24%	27%	29%	32%	34%	36%	39%	40%	44%	43%	47%	45%	47%	46%	47%	48%	48%
dfg1	0%	9%	0%	12%	0%	15%	0%	18%	0%	21%	0%	24%	2%	24%	4%	24%	7%	25%	9%	25%	11%	25%
dfg2	0%	16%	6%	0%	12%	8%	18%	23%	23%	29%	27%	33%	30%	37%	35%	40%	41%	38%	47%	37%	53%	36%
dfg3	27%	27%	34%	37%	41%	47%	46%	53%	49%	55%	52%	57%	53%	58%	54%	59%	56%	59%	57%	59%	58%	59%
dfg4	39%	41%	37%	42%	34%	44%	35%	45%	40%	47%	44%	49%	44%	51%	44%	53%	44%	53%	44%	52%	44%	52%
dfg5	28%	40%	34%	41%	39%	42%	42%	44%	44%	46%	47%	47%	50%	49%	50%	49%	50%	50%	50%	50%	50%	50%
dfg6	25%	26%	30%	35%	35%	43%	41%	49%	48%	51%	54%	54%	52%	55%	51%	56%	51%	57%	52%	58%	54%	59%
dfg7	0%	0%	13%	15%	26%	30%	36%	40%	43%	46%	50%	52%	50%	53%	50%	55%	51%	56%	54%	57%	56%	58%
Average	**16%**	**18%**	**21%**	**25%**	**26%**	**32%**	**31%**	**37%**	**35%**	**40%**	**38%**	**43%**	**40%**	**45%**	**42%**	**47%**	**44%**	**48%**	**46%**	**49%**	**47%**	**49%**

Normalized Latency Constraint (×)

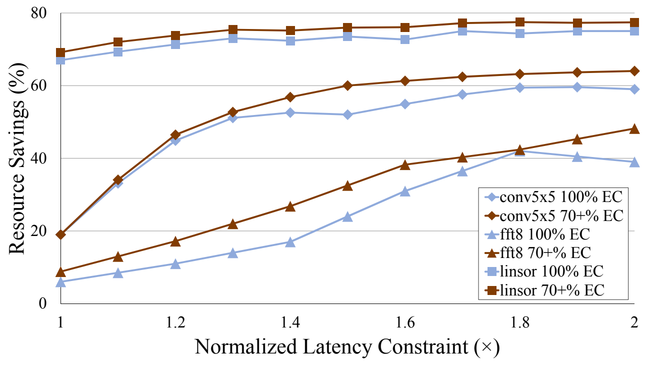


Fig. 6. Trends and exceptions observed on comparison with TMR applied to existing RTL. A common trend seen is diminishing returns with increase in latency (e.g., conv5x5 benchmark). An exception to this is observed for the linsor benchmark, where dependencies in the DFG enable large resource savings that are not apparent to an RTL design. Consistent resource savings improvement with relaxation in the error correction constraint is also commonly observed, as seen in the fft8 benchmark results.

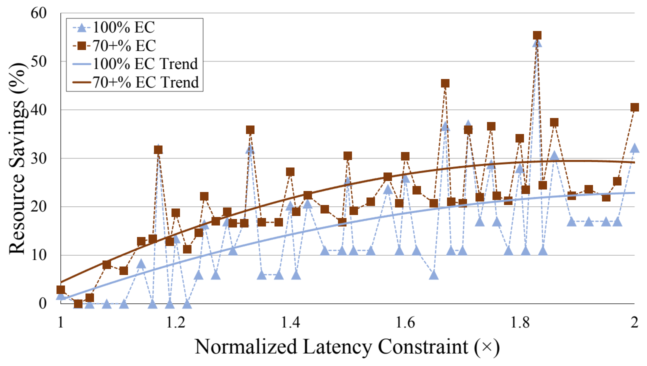


Fig. 7. Summary of FTA Heuristic's resource savings with respect to the optimized RTL w/ TMR. The lighter lines represent results where the error correction was 100%. The darker lines represent the average of results where the error correction was varied between 70% and 99%. The dashed lines represent the average observed for all the benchmarks combined, while the solid lines represent trend-lines approximated as 2^{nd} order polynomials.

One clear trend was that resource savings experienced diminishing returns with increased latency constraints. Fig. 6 demonstrates an example for 5×5 convolution benchmark (conv5x5), where improvements became much less significant after an increased latency constraint of 1.5×. For designs that are not tightly latency constrained, this result provides an attractive tradeoff where designers may be able to sacrifice latency to achieve reduction in resources, while still meeting error constraints. The majority of other benchmarks also exhibit this trend. An exception is observed in the resource savings for the SOR linear equation solver benchmark (linsor), seen in Fig. 6. These increased savings were due to the dependencies between sections of the DFG, which enabled large amounts of resource sharing that may not be immediately apparent to an RTL design. Averaged over latencies of 1× to 2×, the linsor benchmark experienced a significant 73% average resource savings for an error constraint of 100% and 75% average resource savings when the error constraint was lowered between 70% and 99%.

Another interesting trend was that in most cases, there was a consistent increase in the resource savings when the error constraint was slightly lowered. For example, the fft8 benchmark shown in Fig. 6 achieved average resource savings for an error constraint ranging between 70% and 99% that were up to 9.8% higher than an error constraint of 100%.

C. Comparison with RTL optimized for TMR

This section compares the FTA heuristic with RTL that a designer optimizes for TMR. Due to space constraints, we omit a table with all the benchmarks and instead provide the summary shown in Fig. 7. Although the average resource savings here were lower than the previous experiments, these results represent the minimum savings that the heuristic will be able to achieve compared to an RTL implementation. Even though the heuristic showed 0% savings for 100% error correction under some latency constraints, in some cases the heuristic achieved an improvement of up to 58% for 100% error correction. For an error constraint above 70%, the heuristic achieved savings up to 61%. Moreover, even when decreasing the error constraint to as low as 50%, the heuristic still found results with minimum resource usage but with a much higher EC% of up to 98%.

VI. Conclusions

In this paper, we introduced the Fault-Tolerance-Aware Heuristic to solve the minimum-resource, latency- and error-constrained scheduling and binding problem. This heuristic provides attractive tradeoffs compared to TMR by performing redundant operations on shared resources. Unlike TMR, which triplicates all resources, our approach enables a designer to specify a desired amount of error correction, which the heuristic meets while using the minimum number of resources. Experimental results show typical resource savings ranging from 16% to 49% when compared to TMR applied to existing RTL code. When compared to RTL that is optimized for fault tolerance before performing TMR, the heuristic was able to save more than 30% of resources for some examples. Future work includes support for pipelined circuits, optimizations that minimize glue logic introduced by resource sharing, and optimizations to minimize FPGA resources as opposed to the total number of arithmetic resources.

Acknowledgment

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

References

[1] H. Quinn, D. Roussel-Dupre, M. Caffrey, P. Graham, M. Wirthlin, K. Morgan, A. Salazar, T. Nelson, W. Howes, E. Johnson, J. Johnson, B. Pratt, N. Rollins, and J. Krone, "The cibola flight experiment," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, pp. 3:1–3:22, Mar. 2015.

[2] 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*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 249–258.

[3] A. Orailoglu and R. Karri, "A design methodology for the high-level synthesis of fault-tolerant asics," in *VLSI Signal Processing, V, 1992., [Workshop on]*, Oct 1992, pp. 417–426.

[4] ——, "Automatic synthesis of self-recovering VLSI systems," *Computers, IEEE Transactions on*, vol. 45, no. 2, pp. 131–142, 1996.

[5] K. Kyriakoulakos and D. Pnevmatikatos, "A novel sram-based fpga architecture for efficient tmr fault tolerance support," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Aug 2009, pp. 193–198.

[6] T. Inoue, H. Henmi, Y. Yoshikawa, and H. Ichihara, "High-level synthesis for multi-cycle transient fault tolerant datapaths," in *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*. IEEE, 2011, pp. 13–18.

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

[8] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 5, no. 3, pp. 379–395, 1986.

[9] P. G. Paulin and J. P. Knight, "Scheduling and binding algorithms for high-level synthesis," in *Design Automation, 1989. 26th Conference on*. IEEE, 1989, pp. 1–6.

[10] S. Safari, "Co-evolutionary reliability-oriented high-level synthesis," in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, May 2008, pp. 2026–2029.

[11] C. Bolchini and A. Miele, "Design space exploration for the design of reliable sram-based fpga systems," in *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS'08. IEEE International Symposium on*. IEEE, 2008, pp. 332–340.

[12] A. Sengupta and S. Bhadauria, "Bacterial foraging driven exploration of multi cycle fault tolerant datapath based on power-performance tradeoff in high level synthesis," *Expert Systems with Applications*, 2015.

[13] G. Buonanno, M. Pugassi, and M. Sami, "A high-level synthesis approach to design of fault-tolerant systems," in *VLSI Test Symposium, 1997., 15th IEEE*. IEEE, 1997, pp. 356–361.

[14] K. Morgan, D. McMurtrey, B. Pratt, and M. Wirthlin, "A comparison of tmr with alternative fault-tolerant design techniques for fpgas," *Nuclear Science, IEEE Transactions on*, vol. 54, no. 6, pp. 2065–2072, Dec 2007.

[15] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie, "Reliability-centric high-level synthesis," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 1258–1263 Vol. 2.

[16] E. Stott, P. Sedcole, and P. Cheung, "Fault tolerant methods for reliability in fpgas," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept 2008, pp. 415–420.

[17] A. Ben Dhia, L. Naviner, and P. Matherat, "A new fault-tolerant architecture for clbs in sram-based fpgas," in *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, Dec 2012, pp. 761–764.

[18] C. Bolchini, D. Quarta, and M. D. Santambrogio, "Seu mitigation for sram-based fpgas through dynamic partial reconfiguration," in *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '07. New York, NY, USA: ACM, 2007, pp. 55–60.

[19] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant fpga systems," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, no. 2, pp. 212–221, June 1998.

[20] K. Nikolic, A. Sadek, and M. Forshaw, "Fault-tolerant techniques for nanocomputers," *Nanotechnology*, vol. 13, no. 3, p. 357, 2002.

[21] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin, "Improving fpga design robustness with partial tmr," in *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, March 2006, pp. 226–232.

[22] B. Pratt, M. Caffrey, J. Carroll, P. Graham, K. Morgan, and M. Wirthlin, "Fine-grain seu mitigation for fpgas using partial tmr," in *Radiation and Its Effects on Components and Systems, 2007. RADECS 2007. 9th European Conference on*, Sept 2007, pp. 1–8.

[23] K. Siozios, D. Soudris, and M. Hübner, "A framework for supporting adaptive fault-tolerant solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 5s, pp. 169:1–169:22, Dec. 2014.

[24] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, "On the optimal design of triple modular redundancy logic for sram-based fpgas," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*. IEEE Computer Society, 2005, pp. 1290–1295.

[25] J. Vite-Frias, R. Romero-Troncoso, and A. Ordaz-Moreno, "Vhdl core for 1024-point radix-4 fft computation," in *Reconfigurable Computing and FPGAs, 2005. ReConFig 2005. International Conference on*, Sept 2005, pp. 4 pp.–24.

[26] J. Hu, "Solution of partial differential equations using reconfigurable computing," Ph.D. dissertation, The University of Birmingham, 2010.

[27] C. Brandolese, W. Fornaciari, and F. Salice, "An area estimation methodology for fpga based designs at systemc-level," in *Design Automation Conference, 2004. Proceedings. 41st*, July 2004, pp. 129–132.